ISSN 004-8917

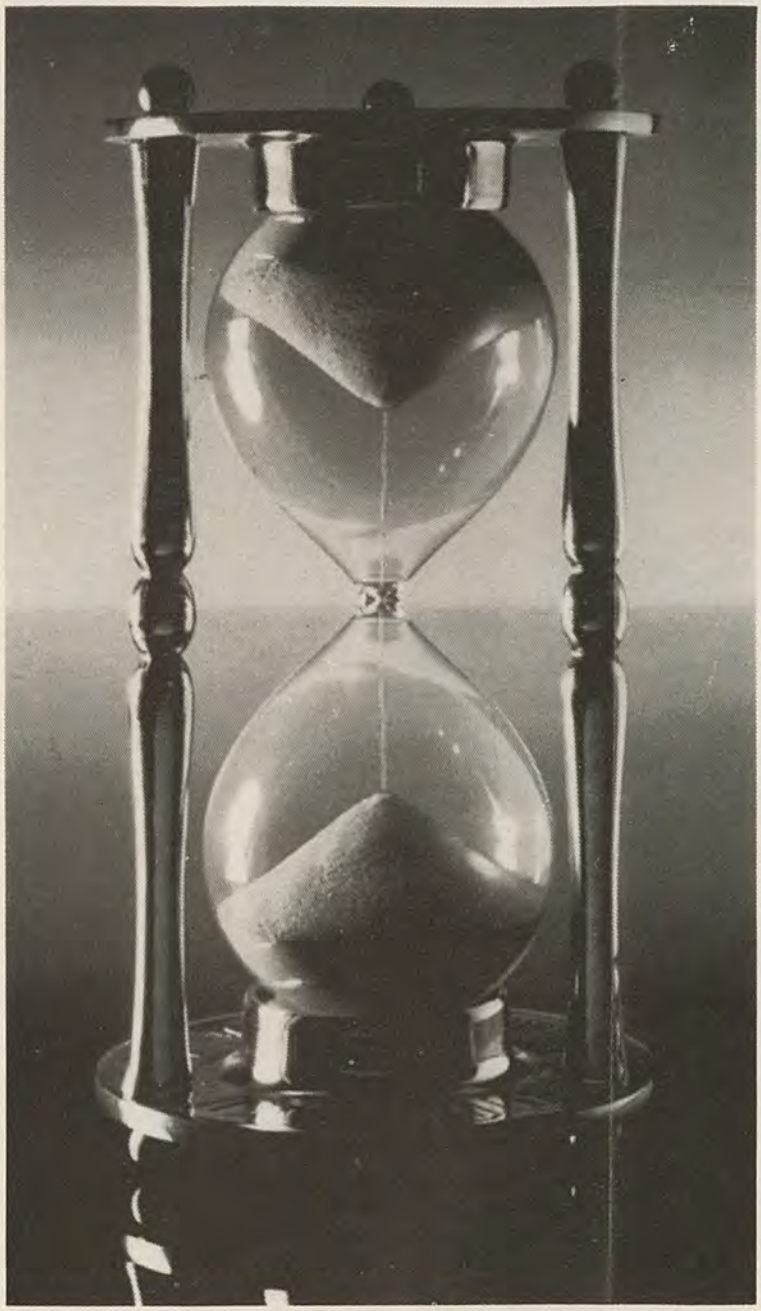
THE AUSTRALIAN COMPUTER JOURNAL

VOLUME 15, NO. 2, MAY 1983

CONTENTS

Published for Australian Computer Society Incorporated
Registered by Australia Post, Publication No. NBG 1124

We've turned the world of computer technology upside down.



Until Tandem(TM) came along, you had to live with the very real threat of your computer going down, corrupting or losing data and generally causing havoc. An especially grim prospect if yours is an on-line operation.

Now Tandem's NonStop(TM) technology has put an end to all that.

For NonStop offers the highest degree of fault tolerance on the market today for large volume on-line transaction processing applications.

Should a component fail, no operator intervention is required. NonStop's operating software takes over immediately without any interruption whatsoever to system availability.

Complete data integrity is assured, even if the data base is geographically distributed.

The **Relational** data base can be distributed over multiple disc volumes and transaction can execute in up to 4,080 processors in a distributed network.

If you find all this too good to believe, phone Liz Penny on (03) 267 1577. She'll invite you to spend some time with us to put Tandem NonStop through its paces.

But be warned, demonstrations like this have already convinced others of NonStop's unique capabilities.

So probably we'll turn your thinking upside down too.

TANDEM NonStop™ Systems

For further information contact:

TANDEM NONSTOP PTY. LTD.

3 Bowen Crescent,
Melbourne, Vic. 3004
Telephone: (03) 267 1577

22 Atchison Street,
St. Leonards, N.S.W. 2065
Telephone: (02) 438 4566

S.G.I.O. Building,
Cnr. Turbot & Albert Sts,
Brisbane, Queensland 4000
Telephone: (07) 229 3766

3rd Floor,
151 South Terrace,
Adelaide, S.A. 5000
Telephone: (08) 212 1110

NCATC/10

News Briefs from the Computer World

"News Briefs from the Computer World" is a regular feature which covers local and overseas developments in the computer industry including new products, interesting techniques, newsworthy projects and other topical events of interest.

ONYX DEALER SEMINAR

Onyx (Australia) Pty. Limited recently held a dealer seminar at The Shore Inn in Sydney on 10 March 1983.

Title of the seminar was "Selling Onyx Solutions" and featured guest speakers Mr Tom Reynolds, Vice President of International Marketing for Onyx Systems, Inc. of California, who gave a talk on the corporate background and philosophies of Onyx; and Miss Bernadette Luongo, Marketing Support Officer for Onyx Systems, Inc., who gave a presentation on the complete range of Onyx products and software.

Other speakers included Mr Bruce Paul from STC and Mr Anthony McAuslan, National Marketing Manager for Onyx (Australia) Pty Limited.

The seminar also included a mini trade fair with third-party companies providing peripherals and software packages now available on the Onyx range.

During the seminar Onyx (Australia) Pty. Limited was presented with an award from Onyx Systems, Inc. for being named the top Onyx distributor outside the USA for its outstanding sales efforts in 1982.

In summing up the seminar, National Marketing Manager for Onyx (Australia) Pty. Limited, Mr Anthony McAuslan, told guests that: "As end-users become better educated, they will look to the computer professional to supply them with a total solution including hardware, software, communications and support and not just a box of nuts and bolts as is sometimes the current practice".

The seminar was attended by over 100 people, representing Onyx dealer organisations from all over Australia and New Zealand.

PERKIN-ELMER APPOINTMENTS

Reacting to continuing changes in the marketplace, Perkin-Elmer Computers announce a number of new appointments and changes in staff responsibilities.

Andy Hirons, previously Northern Region Manager, has been promoted to the position of Australia/New Zealand Vertical Sales Manager with responsibilities for OEM's, Third Parties, Consultants and Application Software.

Andy has been succeeded by Vic Whiteley who joins Perkin-Elmer from General Electric Information Services. Prior to this Vic was employed by Honeywell in a number of management roles.

Keith Reynolds has assumed the position of Southern Region Manager, Keith transferred from Perkin-Elmer UK where he held the position of Northern Region Manager.

INDIVIDUALLY ADJUSTABLE COMPUTER TABLES

To ensure maximum comfort and convenience when working with computers and visual display units, the tables on which they stand must be adjustable. A Dutch manufacturer of office furniture has developed a special range of such tables under the name CTT.

The leaves of this table for the display unit and the keyboard are individually and continuously adjustable in height. The height of the keyboard leaf is adjustable from 50 to 77 cm and that of the display leaf from 65 to 83 cm.

Both leaves are 80 cm wide and 46 cm deep.

The tables come with a cable duct for the display unit cable and the mains power supply. The lifting mechanism is concealed in the central panel.

The computer tables are available in three models, with mechanical, electrical and electronic height adjustment.

In the model with mechanical adjustment the display unit and keyboard leaves are balanced by weights. When the brake is disengaged the heights of the two leaves can easily be adjusted by hand.

Operating buttons control the height adjustment by means of an electrically powered screwed spindle in the electrical model.

(Continued on Page IV at back)

Buy genuine Onyx. For an affordable price.

The genuine onyx stone has qualities that are rare and enduring. The Onyx reputation as a world leader in microcomputer technology is a rare and enduring one. The Onyx/IMI, Incorporated Winchester disk drive is recognised as without parallel. So too is the competitive price structure of Onyx microcomputers.

Genuine Onyx.

Effective translation of all these benefits to information managers in business and industry can really only be achieved by Onyx, the originators. And the only genuine marketing and distribution organisation in Australia is Onyx Australia.

Only Onyx Australia has the intrinsic component knowledge, the specialist technical intelligence and the genuine operating skills to fully maximise the benefits of Onyx microcomputers to users.

Onyx dealer network.

These benefits, this enduring Onyx service is provided to users through a national network of Onyx dealers. Only Onyx and its dealers have total command of the many flexible and rewarding options available to users through extensive Onyx software applications programs.

And through STC users have an Australia wide hardware maintenance facility readily at hand for Onyx servicing.

The affordable Onyx.

The Onyx C5000/8000 series computers and the remarkable Sundance desk top computer all include the Onyx/IMI, Incorporated Winchester disk drive. This disk system enables Onyx computers to perform faster and to store more data, more reliably than comparable or bigger name systems with 'floppy' disks.

But let's not forget price. The functions, the scope of performance (6MB to 160MB range), the reliability of Onyx microcomputer systems and their capacity to grow with the user can all be purchased for much less than the cost of comparable computers.

For better business productivity, profit and greater operations efficiency talk to the source. The genuine Onyx people, Onyx Australia.

Onyx dealers are located in every Australian capital city and in major country centres.

For further information about Onyx microcomputers mail this coupon to:

ONYX

Onyx Australia
Pty Limited

7-9 Merriwa Street
Gordon, N.S.W. 2072.
Telephone: (02) 498 6611

Name

Address

Postcode Phone

Company ☐ Please tick if you are a Computer Dealer

Editorial

At present much attention is being focussed within universities and colleges of advanced education on the teaching of Computer Science, or Computing Science, as some people now prefer to call it. Enrolments in this subject, which hardly existed 15 years ago, now surpass to a significant degree the enrolments in many traditional scientific disciplines such as chemistry and physics. All departments that teach Computing Science have limited resources, and many have had to introduce quite strict quotas in order that their resources not be completely saturated. With increasing frequency, in the race for places in these quotas, Australian students are being overtaken by overseas students. The level at which these quotas are set is thus a matter for serious concern. There have been several occasions when demand for students with particular training (for example, for geologists in the early 1970's) has risen quickly, and then just as quickly has subsided again. From this experience, university and college administrations have become reluctant to redeploy resources and raise quotas too quickly. This reluctance has been compounded recently by the none-too-subtle financial squeeze forced on the whole educational system by the previous federal government. The incoming government seems to be more favourably disposed, but it is going to have great difficulty in sorting out its financial priorities. Clearly, all ACS members who believe that their profession has a firm intellectual foundation that is best imparted via a rigorous formal training should do whatever is in their power to encourage the government in the right direction.

But what should this formal training consist of? One of the challenges that faces universities and colleges of advanced education today is that many members of the computing profession do not support and approve present educational programs in computing. Many members entered the profession at a time when its intellectual foundations were still being carved out, and were not well understood, when there were no formal courses and practitioners had to learn important truths the hard way. Many have had successful careers without ever mastering many of the formal results which now form part of the standard curriculum. From this experience (or lack of it) many of today's practicing professionals incline to the view that their way is the only way, and that most of the material taught in academic courses is irrelevant and/or worse than useless. This is a dangerous and unworthy assumption. I cannot share the view that what the world really needs is more programmers skilled in IBM assembler and RPG. Although there may still be a demand for such people, to equip today's new entrants into the computing profession with these as their primary skills would be a dereliction of duty. (Besides, which educational institutions in today's economic climate can afford to teach programming using one of IBM's large systems?) At my university, we aim to provide Computer Science students with knowledge and skills that will still be useful into the 1990's and beyond. How valid is our view of the future? We began teaching students about Pascal and the UNIX™ timesharing system in the mid-1970's long before we could prove that Pascal would one day be more widely available than Fortran, or that UNIX would cease to be just "a computer scientist's plaything" and become a *de facto* standard operating system by the mid-1980's. With so much former irrelevance suddenly becoming relevant, what could we do for an encore?

For the last two years, several universities including mine, have been teaching some of their students about a non-procedural language called PROLOG, which originated in France about ten years ago and which is based on theorem proving and the predicate calculus. Since probably less than 1% of ACS membership has ever studied the predicate calculus formally, how, you may ask, could such a language ever replace those old favourites, COBOL and BASIC? But such a question is not so far-fetched and maybe the world will not always owe COBOL programmers a living. It is upon developments such as PROLOG that the Japanese are basing the strategy for their fifth generation computer systems, and we all know what paper tigers they have proved to be. But now the good news: you need not be completely taken by surprise because this issue of the Journal contains three tutorial articles relating to PROLOG: an introduction to the language, an introduction to the application of PROLOG to database management, and finally, a description of the basic principles behind one working PROLOG interpreter. Please read them; one day they may be more relevant than you once thought.

PROLOG: A Tutorial Introduction

R. A. Sammut† and C. A. Sammut*

PROLOG is a programming language based on predicate logic. Since its first implementation approximately ten years ago, it has found applications in a variety of "symbol processing" areas such as natural language processing, deductive information retrieval, compiler writing, symbolic algebra, computer-aided design and robot problem-solving.

This paper introduces the fundamental concepts which are unique to programming in PROLOG by developing and analyzing a series of small programs for deductive information retrieval, the solution of the "N-queens" problem and a simple exercise in computer-aided design.

Keywords and phrases: PROLOG, logic programming, deductive databases, artificial intelligence, computer-aided design.

CR categories: A.1, D.3, I.1, I.2, J.6.

1. INTRODUCTION

Approximately ten years ago, the concept emerged of using predicate logic as a programming language (Kowalski, 1974). PROLOG is a language which realizes that concept in practice and which is currently being applied in a variety of areas such as natural language processing (Colmerauer, 1982), deductive information retrieval (van Emden, 1978; Santane-Toth and Szeredi, 1982), compiler writing (Warren, 1980), symbolic algebra (Bergman and Kanoui, 1973), computer-aided design (Markusz, 1977) and robot problem-solving (Warren, 1976). It has also been chosen as the basic programming language for Japan's "Fifth Generation Computer Systems" project (Warren, 1982).

The features of PROLOG which have led to its rapid growth in popularity for such "symbol processing" applications are:

- the very powerful pattern-matching facilities of the language (which replace the "assignment" in conventional languages as the basic underlying operation).
- the backtracking facility which enables a procedure to generate, automatically, a sequence of alternative solutions to a given problem.
- the facts that data structures in PROLOG are general trees and that a procedure call and a data object have exactly the same form.
- the flexibility of PROLOG procedures in that which arguments are "input" and which are "output" is not determined in advance but may vary from one call to the next. PROLOG procedures may also return as output "incomplete" data structures containing variables which have not been "instantiated" (i.e. their values have not been specified). The "blanks" can be filled by subsequent procedures.
- the relative ease with which programs can be written in PROLOG once the initial unfamiliarity of the preceding four features is overcome.

This last "feature" follows essentially from the structure of a PROLOG program. In most programming languages, a program is a description of an algorithm. Collectively, the statements of the program represent the steps required to achieve some specific goal but, individually, they have no meaning. In execution, a PROLOG program can also be viewed in this "procedural" way. However, in writing a PROLOG program, the programmer does not actually specify a sequence of steps. A PROLOG program consists of a set of statements (facts or rules) about objects and their logical relationships. Each statement is essentially self-contained and can be understood independently of the rest of the program. (This is achieved using the *declarative semantics*). The program is correct if each statement is true.

The aim of this paper is not to discuss these features of PROLOG in an abstract way but to demonstrate them with the help of specific sample programs. Each of the examples to be discussed in the following sections is relatively simple but, at the same time, provides an insight into the power of the language and a hint of the variety of "real" applications for which PROLOG is a very useful tool. Before looking at these examples, however, we must introduce some terminology.

2. SUMMARY OF SYNTAX AND TERMINOLOGY

A PROLOG program comprises a set of *procedures*, each of which defines a particular logical relationship, or *predicate*. A procedure consists of one or more assertions, or *clauses*, of the form

```
P0 :- P1, P2, ... Pn.
```

which can be read either declaratively as

"P0 is true if P1 and P2 and ... Pn are true"

or procedurally as

"to satisfy goal P0, satisfy goal P1 then P2 then ... Pn".

(Note that a period terminates every PROLOG clause.) In this clause, P0 is called the *head* goal and the conjunction of goals P1, P2, ... Pn form the *body*. A clause without a body, i.e. a clause of the form

```
P.
```

is a unit clause or a *fact* and means

"P is true" or "goal P is satisfied".

A clause without a head such as

```
:- P1, P2, ... Pn.
```

is a *directive* and is the means by which a PROLOG pro-

†School of Electrical Engineering and Computer Science, University of New South Wales, Kensington, NSW 2033. *Present address: Department of Mathematics and Computer Science, St. Joseph's University, Philadelphia, PA 19131 USA. Manuscript received February 1983, revised April 1983.

gram is invoked to execute goals P1, P2, . . . Pn. The directive is interpreted as either

"Are P1 and P2 and . . . Pn true ?"

or

"Satisfy goal P1 then P2 then . . . Pn".

A special form of directive which is commonly used is the *question*, written as

? P1, P2, . . . Pn.

The difference between questions and other directives will be explained shortly.

Before proceeding with this abstract discussion, however, let us have a look at some specific examples of PROLOG clauses. The following are two clauses from a procedure defining a predicate *studies* which associates students and the subjects they study:

```
studies(jack, 620).
studies(jill, 641).
```

The first clause may be read as "jack studies (subject) 620" and the second as "jill studies (subject) 641". To these simple clauses we can add a more general clause of the form

```
studies(Student, 611) :- year(Student, 1).
```

which may be read as either

"any person, Student, studies subject 611 if that person is in year 1"

or

"to find a person who studies 611, find a person who is in year 1".

These three clauses contain examples of every type of data object found in PROLOG. The clauses, their component goals and the arguments of those goals are called *terms*. In general a term is either a constant, a variable or a compound term (or structure):

Constants are definite objects, like proper nouns in natural language, and can be either integers (611, 1, 620) or *atoms (studies, jack, :-)*.

Variables are distinguished by an initial capital letter (*Student*) or by the initial character "_". They do not represent storage locations as in most programming languages but are local names for some specific (but undefined) object. Two variables with the same name in different clauses are completely distinct.

Compound terms are structures like *studies(jack, 620)* or *year(Student, 1)*. The compound term *studies(jack, 620)* is said to have *functor*, *studies* and *arguments*, *jack* and *620*. All data structures in PROLOG are compound terms although, as we shall see shortly, a special notation has been provided for lists.

The basic operation underlying a PROLOG program is the *matching* or *unification* of terms. Two terms match if their functors and all arguments match. Moreover, if any of the arguments are variables which have not been bound to a constant term (i.e. they are *uninstantiated*) then they will match any other argument in the same position. For example, *studies(jack, 620)* will not match *studies(jill, 641)* but will match *studies(X, 620)*. When this match is made, the variable *X* is bound to the constant value *jack*.

We now have a sufficiently large vocabulary to proceed to look at an example of a PROLOG program.

3. A SAMPLE PROGRAM: DEDUCTIVE INFORMATION RETRIEVAL

The three clauses for *studies* above are part of a procedure which asserts the subjects studied by students in a University department. This procedure, in turn, is part of a larger program which contains information about lecturers teaching various subjects, students' course years and timetabling information such as the time and location of various classes. In reality, such a program could be very large so, for the sake of illustration, we abbreviate it:

```
lectures(jeff, 611).
lectures(ken, 620).
lectures(david, 641).
lectures(ian, 642).
lectures(ken, 643).
lectures(graham, 646).

studies(fred, 611).
studies(jack, 620).
studies(jill, 641).
studies(jill, 646).
studies(henry, 643).
studies(henry, 646).
studies(X, 611) :- year(X, 1).

year(fred, 1).
year(jack, 2).
year(jill, 3).
year(henry, 3).

class(611, m1000, lg1).
class(611, w1300, lg1).
class(646, tu1100, g24).
class(622, th1000, 418).
class(643, tu1100, 224).
```

In this form, the program is essentially just a database. Notice that each clause is completely self-contained. That is, the program does not consist of a sequence of steps in an algorithm as in a conventional language. We simply have a collection of assertions of facts or logical relationships and we would still have a valid program if clauses or whole procedures were interchanged. (The actual result of execution of a particular program might change if clauses within a procedure were re-ordered but the reason for this will become clearer later).

How do we actually use this program? The simplest thing we can do is to ask a question such as, "Does jill study subject 641?":

```
? studies(jill, 641).
```

PROLOG responds to this question by writing *true* if the given goal is true within the context of this program. To find whether the goal *is* true, PROLOG tries to match it with the head of a clause in the program (by searching from the top of the program). If a match is found, the matching clause is then *activated* by executing, in turn, each of the goals (if any) in its body. The head goal is true if each of the goals in the body is also true.

Thus in the present example, PROLOG looks through the procedure for *studies* and finds that the given goal does match one of the clauses in the procedure. Since the body of the clause is empty, the goal is automatically satisfied. We would, therefore, receive the response:

```
true
```

In this form, we are only using the program to check information about specific objects. A much more useful application would be, for example to ask PROLOG to find all students in year 3:

```
? year(Student, 3).
```

Here *Student* is a variable and PROLOG responds by generating all instances of the variable for which the goal is true. It does this by matching the goal against clauses in the procedure *year*. An uninstantiated variable matches against anything so that the first match will be with the clause

year(jill, 3). PROLOG therefore responds

```
Student = jill
```

indicating that the goal is true if *Student* is instantiated to *jill*. If at any time PROLOG fails to find a match for a goal, it *backtracks*, i.e. it rejects the most recently activated clause, undoing any substitutions made by the match with the head of the clause. It then reconsiders the original goal which activated the rejected clause and tries to find a subsequent clause which also matches the goal. This same procedure also applies when PROLOG has successfully found one solution to a question. It automatically backtracks to look for alternative solutions. So, in this example, PROLOG "undoes" the binding of *Student* to *jill* and looks for an alternative solution. On doing this, it finds that the only other match occurs when

```
Student = henry
```

Before going any further, we shall pause to explain the difference between the question we have just asked and the directive

```
:– year(Student, 3).
```

In both cases, we are asking PROLOG whether *year(Student, 3)* is true for any *Student* or, equivalently, to find a value of *Student* such that *year(Student, 3)* is true. In the case of the directive, PROLOG finds a value of *Student*, if one exists, but produces no output and does not backtrack to find alternative values which might also satisfy the goal. If we want to print out the value of *Student*, we need to explicitly use a separate (built-in) predicate to do so:

```
:– year(Student, 3), print(Student).
```

Moreover, if we want to find all possible values of *Student*, we have to force backtracking to occur by making the goal fail after writing the first value. PROLOG will then attempt to re-satisfy *year(Student, 3)*. We do this by using another built-in predicate called, appropriately, *fail:*

```
:– year(Student, 3), print(Student), fail.
```

When we use a question mark at the beginning of a directive we are thus automatically asking for all variable values to be printed and for backtracking to find all possible solutions.

Now let us return to our program and ask some slightly more complicated questions. Suppose we want to know whether *jill* and *henry* study any subjects in common. We ask

```
? studies(jill, X), studies(henry, X).
```

PROLOG attempts to satisfy goals from left to right so it first finds an instance of the variable *X* (i.e. a subject) for which *studies(jill, X)* is true. The first match occurs with *X* instantiated to *641* so the first goal is satisfied and the value *641* is substituted for the variable *X* throughout the clause. We then go on to try to satisfy the second goal which has become *studies(henry, 641)*. Obviously this clause does not appear in the program so this goal fails. PROLOG then goes back to the most recently executed goal (in this case, there is only one preceding goal) and seeks an alternative match. That is, it undoes the instantiation of *X* with *641* and looks for an alternative match for *studies(jill, X)*. The next alternative occurs with *X* replaced by *646* and the second goal is satisfied when this value is substituted for *X*. The complete goal is therefore satisfied and PROLOG outputs

```
X = 646
```

Having found one solution, PROLOG backtracks again to look for other possible solutions. In this case there are none so execution terminates.

Deducing complex relationships

We could similarly go on to ask for all students studying a particular subject, all subjects taught by a particular lecturer, etc. However we are not restricted to asking such simple questions. PROLOG can be used to deduce new relationships between objects. For example, suppose we want to find all the students taught by a lecturer. We could do this by adding to the program a predicate called *teaches* defined by

```
teaches(Lecturer, Student) :–
        lectures(Lecturer, Subject),
        studies(Student, Subject).
```

which means that *Lecturer* teaches *Student* if *Lecturer* lectures *Subject* and *Student* studies *Subject*. Then if we ask

```
? teaches(ken, Student).
```

the response will be

```
Student = jack
Student = henry
```

Similarly, the question

```
? teaches(X, jill)
```

would result in

```
X = david
X = graham
```

(Note that in the first question above, the first argument, *ken*, was input and the second argument returned output. In the second question, the roles of the two arguments are reversed. We could go a step further and ask *? teaches(X, Y).* to find all teacher-student pairs.)

If we were using this database in order to design or modify a timetable, we would want to know whether there were any clashes, i.e. whether a student were ever required to be in two classes at the same time or whether a room was booked for two different classes at the same time. To find such clashes, we could add to the program a procedure such as

```
clash(Room, S1, S2, T) :–
                class(S1, T, Room),
                class(S2, T, Room),
                S1 /= S2.
clash(Student, S1, S2, T) :–
                studies(Student, S1),
                class(S1, T, _),
                studies(Student, S2),
                S1 /= S2,
                class(S2, T, _).
```

In this procedure, *S1 /= S2* is an infix form of the goal */=(S1, S2)*. This is a built-in predicate which is true if the variables *S1* and *S2* are instantiated to either integers or atoms and the two have different values. (A predicate written in infix form like this is called an *operator*. All the usual comparison operators (=, /=, <, >, <=, >=) are available as built-in predicates in PROLOG.)

The first clause says that a clash occurs in room *Room* at time *T* if subject *S1* takes place in *Room* at that time, subject *S2* also takes place in *Room* at the same time and the two subjects are not the same. The second clause says that *Student* has a clash if he studies subject *S1* which has a class at time *T* and he also studies a different subject *S2* which has a class at the same time. Note that in this case, we don't care in which rooms these classes occur so PROLOG allows us to save on variable names by using the anonymous variable "_" as the third argument of the two *class* goals. This anonymous variable differs from all other variables in that two instances of "_" within the same clause do not share the same value.

Asking the question, "Are there any clashes of rooms or students?"

```
? clash(X, S1, S2, T).
```

would result in

```
X = henry
S1 = 643
S2 = 646
T = tu1100

X = henry
S1 = 646
S2 = 643
T = tu1100
```

which tells us that a clash involving the student *henry* will occur on Tuesday at 11 am (tu1100). Observe that PROLOG gives us essentially the same solution twice. We leave it to the reader to explain why this occurs.

Collecting terms in lists

Until now, when we have asked a question such as, ? *year(X, 3).* each value of X has been produced separately and we have no way to manipulate the collection of all instances of X satisfying the goal. For example, we might want to arrange a group of students in alphabetical order or to count how many students satisfy a certain goal.

An obvious way to group the various solutions together is to form a *list*. A list can be either the atom "[]" representing the empty list or a compound term with a functor (let's call it *list*) and two arguments which are, respectively, the *head* and *tail* of the list. Thus a list containing the first letter of the alphabet would be represented by the structure

```
list(a, [])
```

and a list of the first three letters would be

```
list(a, list(b, list(c, []))).
```

To simplify manipulations with lists, PROLOG provides a special notation. In this notation, the last structure can be abbreviated to

```
[a, b, c]
```

and a list whose tail is a variable is written as

```
[a, . .L]    or    [a, b, . .L]
```

where, in the second case, *a* and *b* are the first two elements of the list and *L* is the tail.

Note that this special notation is purely for convenience – the internal structure of the list within PROLOG is just like any other compound term with a functor and two arguments.

Now suppose we want to make a list of all the lecturers in this department or of all the students who share some particular property. Our PROLOG implementation contains a built-in predicate called *findall* which is defined so that the goal

```
findall(X, P, L)
```

constructs a list L consisting of all values of the variable X for which the goal P is satisfied. The elements are listed in the order in which they are found. (*findall* is not a "first-order predicate" in that it requires changes to be made to the program's database during execution. This can be done in PROLOG using the *assert* and *retract* predicates [see, for example, Clocksin and Mellish, 1981] but is beyond the scope of this paper.)

Thus to list all the students in year 3, we would type

```
:– findall(X, year(X, 3), L), print(L).
```

and receive the response

```
[jill, henry]
```

Similarly, to write out a list of all lecturers we would say

```
:– findall(X, lectures(X, _), L), print(L).
```

and PROLOG would respond

```
[jeff, ken, david, ian, ken, graham]
```

Note that since we want a list of all lecturers, regardless of their subject, we seek all values of X for which *lectures(X, _)* is satisfied. This will match with all clauses for *lectures* regardless of the subject in the second argument. Note also that because *ken* gives lectures in two subjects, his name appears twice in the list.

A simple sorting procedure

As a final addition to our program before we leave this illustration of PROLOG as a deductive query language, let us now write a procedure to sort a list L into alphabetical order and to remove duplicate entries. We will do this using a simple insertion sort procedure but more sophisticated algorithms such as *quicksort* can be programmed with little extra effort (van Emden, 1977).

We split the sorting algorithm into two procedures – one to *insert* an element in the correct position in a list and a second to actually *sort* the whole list.

Our sort procedure will consist of only two clauses:

```
sort([], []).
sort([H, . .T], S) :–
        sort(T, L),
        insert(H, L, S).
```

where *sort(X, Y)* means that X and Y are lists and Y is a sorted version of X. The first clause simply says that, when sorted, the empty list remains the empty list. The second clause says that we sort the list $[H, . .T]$ by first sorting the tail T to produce the list L and then inserting H in the correct position within L to produce the sorted list S. Thus *sort* calls itself recursively with smaller and smaller lists as its arguments until the goal *sort*([], []) succeeds. It then builds up the sorted list by inserting the elements in their correct order as it works its way back up through the sequence of recursions.

The procedure for inserting an element into a list is equally simple:

```
insert(X, [X, . .T], [X, . .T]).
insert(X, [H, . .T1], [H, . .T2]) :–    X > H,
                                        insert(X, T1, T2).
insert(X, [H, . .T], [X, H, . .T]) :–   X < H.
insert(X, [], [X]).
```

where *insert(X, L1, L2)* means that X is inserted in list $L1$ to produce list $L2$. The first clause deals with the case where the value of X is equal to the head of the list. In that case, we do not wish to insert a duplicate element with this value so the list $[X, . .T]$ is left unchanged. Note that this clause is equivalent to

```
insert(X, [H, . .T], [H, . .T]) :–    X = H.
```

but is much simpler as PROLOG's pattern matching automatically checks whether X is equal to the first element of the list. (The original form is also more desirable as data structures should be exhibited explicitly wherever possible).

The second clauses says that if X is greater than the head of the list then we insert X into $[H, . .T1]$ by first inserting X into $T1$ to produce $T2$ and then forming the list $[H, . .T2]$. The third clause says that if $X < H$ then we insert X at the head of the list. The final clause deals with the trivial case where we insert an element into the empty list.

With these two procedures defined, let us now return to our task of alphabetically sorting the list of lecturers in

our database. To do this, we issue the directive

```
:- findall(X, lectures(X, _), L), sort(L, S), print(S).
```

to which the response is

```
[david, graham, ian, jeff, ken].
```

There are obviously many more elaborations which could be made to a deductive information retrieval system such as this. Practical applications of PROLOG along the lines of this example range from information systems for predicting the properties of various chemical structures (Santane-Toth and Szeredi, 1982) to the possibility of representing the law as a logic program (Sergot, 1982). However, our aim is to introduce the language rather than to delve deeply into database systems. A more detailed discussion of such systems is given by Lloyd (1983). We will now move on to look at a completely different kind of problem for which the logic programming features of PROLOG are also well suited.

4. PROBLEM-SOLVING WITH PROLOG

One of the more commonly discussed puzzles in programming texts is the "N-queens problem". The aim of this problem is to find all the ways of placing N queens on an N x N chess board so that no queen "attacks" another, where two queens are said to attack each other if they are positioned along a common row, column or diagonal.

It is possible to solve this problem using conventional programming languages and a very detailed discussion of how this is done is given by Dijkstra (1972). However, as will be clear either from reading that work or, simply from thinking about the problem for a short time, there is a considerable amount of effort involved in determining the most effective procedure for generating alternative configurations to find a particular solution and for subsequently backtracking to make sure that all possible solutions are found.

In contrast, because it does not need to specify the precise sequence of steps required to solve a problem but only the rules to be followed and the goal to be satisfied, a PROLOG program for the N-queens problem is almost trivial.

Before looking at the program, let us consider the data structures we are going to use to represent the solutions. The two possible solutions for a 4 x 4 board are

```
. . Q .          . Q . .
Q . . .          . . . Q
. . . Q          Q . . .
. Q . .          . . Q .
```

and clearly, we could represent these using the list

```
[2,4,1,3]   and   [3,1,4,2]
```

where the first element gives the column number of the queen in the first row, the second element gives the column number for the second row and so on. For any value of N, all possible solutions must have exactly one queen in each row and column, so lists such as those above are sufficient to describe all solutions. However, in order to be a valid solution, a particular configuration must also satisfy the condition that no two queens lie along the same diagonal (upward (/) or downward(\)). Therefore, as we generate the list of column numbers representing a particular solution we need to be able to check whether any two elements in the list attack each other along their diagonals.

Along an upward diagonal, the difference between row and column numbers is constant; along a downward diagonal, the sum is constant. So each upward and downward diagonal can be uniquely specified by a single integer. The task of checking the validity of a particular configuration of queens can therefore be simplified if we replace the list of column numbers above by a list of terms of the form

```
square(C, U, D)
```

where C characterizes the column, U the upward diagonal and D the downward diagonal. Thus the solutions above would be represented by

```
[square(2,-1,3),square(4,-2,6),square(1,2,4),square(3,1,7)]
```

and

```
[square(3,-2,4),square(1,1,3),square(4,-1,7),square(2,2,6)].
```

This is the representation we use in the program below although we emphasize that, once a solution is found, only the list of column numbers is actually needed to describe the solution. The other arguments in the *square* terms are included only for convenience. But now let's think about how we should choose the squares to put into the list. In other words, how do we actually solve the problem?

If we were to tackle it by hand, this is probably how we would approach the problem for an N x N board:

1. Place a queen somewhere in the first row.
2. Proceed to the next row and place a queen in one of the columns which is still free. Check that this position is not attacked along either diagonal and then repeat step 2 until N queens have been placed on the board.

If, at any point, we place a queen in an unsafe position, then we backtrack to the last choice we made and try another alternative. Similarly, when we have found one complete solution, we backtrack through the various choices we have made to generate all possible solutions.

While this is obviously a long and tedious procedure to follow by hand (particularly for an 8 x 8 board where there happens to be 92 solutions) the actual statement of the rules to be followed is quite simple. The PROLOG program below reflects this simplicity. In fact (and this is where PROLOG wins out over conventional programming languages for application of this type) the program is even simpler than the preceding discussion because no mention needs to be made of backtracking – this happens automatically:

```
solve(Input, Output, [Row, . .R], Columns) :-
        choose(Col, Columns, C),
        Up is    Row - Col,
        Down is Row + Col,
        safe(Up, Down, Input),
        solve([square(Col, Up, Down), . .Input], Output, R, C).
solve(L, L, [], []).

choose(X, [X, . .Y], Y).
choose(X, [H, . .T1], [H, . .T2]) :-
        choose(X, T1, T2).

safe(U1, D1, [square(_, U, D), . .Rest]) :-
        U1 /= U,
        D1 /= D,
        safe(U1, D1, Rest).
safe(_,_, []).
```

For a 4x4 board, for example, we would invoke the program with

```
? solve([], Soln, [1,2,3,4], [1,2,3,4]).
```

To follow the workings of the program, we begin with the predicate *solve*. This goal has four arguments: a list, *Input*, of squares which have already been occupied, a list, *Output*, of squares occupied in the final solution and two further lists which contain the row and column numbers which are still free. Initially, *Input* is the empty list, *Output* is an uninstantiated variable and the other two lists contain

the integers 1, . . .N.

To satisfy the goal *solve*, PROLOG first chooses a column from among those available then, having advanced to the next row, calculates the parameters *Up* and *Down* which characterize the diagonals. (This "calculation" will be explained shortly.) It then checks that this position is safe, inserts it in the list of occupied positions and calls *solve* again to place the next queen.

When, finally, a queen has been placed in each of the N rows, the list *R* in the recursive call to *solve* will be empty. Since [] does not match [*Row*, . .*R*], the call to *solve* will fail to match the head of the first clause but, instead, match with the second. The function of this second clause is to complete the solution by passing its first argument, which has been accumulating the list of safe squares, to the second argument, which returns the final solution. For the case of a 4 x 4 board again, this is

```
Soln = [square(3,1,7), square(1,2,4), square(4,–2,6),
square(2,–1,3)]
```

Backtracking then occurs to produce all possible solutions. (The reader will notice that this list is reversed relative to the solution given in the discussion of data structures above. This has happened because the first row of the board was filled first and subsequent squares have been added to the head of the list. In view of the symmetry of the problem, this reversal is irrelevant but we shall see, at the end of this section, how it can be "corrected".)

In arriving at the solution, *solve* calls on three other procedures: *choose*, *is* (a built-in predicate) and *safe*. The goal *choose(Col, Columns, C)* means that the element *Col* is chosen from the list, *Columns* (containing the numbers of all unoccupied columns) leaving the reduced list, *C*. The first clause of the procedure *choose* picks out the head of the list. However, if the column represented by this element does not correspond to a safe position then PROLOG will backtrack to *choose* and the second clause will be matched. This clause picks out an element from the tail of the list by a recursive call to *choose*. (In section 5, we will examine this procedure in a little more detail to demonstrate how backtracking automatically generates all possible solutions to a question.)

The second and third goals in the body of *solve* which calculate the values of *Up* and *Down* use the bujlt-in predicate *is* and also two built-in arithmetic operators + and – so we need to explain a little about how arithmetic is performed in PROLOG.

Along with the comparison operators discussed in section 3, PROLOG also provides built-in operators for integer addition (+), subtraction (–), multiplication (*), division (/), exponentiation (^), modulus (mod) and unary minus, together with their usual associativity and precedence rules. An expression such as *A * B* is simply a PROLOG structure, **(A, B)* like *year(X, 3)* and is not an instruction to multiply *A* and *B*. However the arithmetic operators are called *evaluable functors* because PROLOG provides a mechanism for actually evaluating the arithmetic expression. This mechanism is the built-in predicate *is*.

If we define an integer expression *X* to be a term built from evaluable functors, integers and variables then the procedure

```
Z is X
```

is interpreted as

> "evaluate the integer expression X and unify (match) the result with Z"

At the time of evaluation, any variables in *X* must be instantiated to integer values or the procedure fails.

After this brief introduction, we leave arithmetic and go on to the final procedure used by *solve*, namely, *safe*. The goal *safe(Up, Down, Input)* means that the square characterized by diagonals *Up* and *Down* is safe if no square in the *Input* list lies on those diagonals. The first clause of *safe* says that the values of *U1* and *D1* are safe if the square at the head of the *Input* list has different values of *U* and *D* and if *U1* and *D1* are also safe from the remaining elements of the list. The second clause says that *safe* is true for any values of *Up* and *Down* if the *Input* list is empty.

This essentially completes the solution of the N-queens problem. However both the input and output of our existing program are rather inconvenient – the input because we have to type in two long lists of numbers and the output because a list like *Soln* above is obviously difficult to visualize as a configuration on a chess board. So to conclude this section, we offer some minor embellishments to the program.

To tackle the input problem, we add the following two predicates:

```
queens(N, Solution) :–
        inlist(N, [], L),
        solve([], Solution, L, L).


inlist(1, L, [1, . .L]).
inlist(X, L1, L2) :–
        X > 1,
        Y is X – 1,
        inlist(Y, [X, . .L1], L2).
```

Given a positive integer value, N, the procedure *inlist* generates a list of the integers 1, . . .N. The question

```
? queens(8, X).
```

will, therefore, first call *inlist* to produce the list [1,2,3,4,5,6,7,8] then satisfy the goal *solve([], X, L, L)* with this list substituted for *L*. The solutions will be returned as the values of *X*.

As for the output, one way to simplify this would be to use the procedure *printlist:*

```
printlist([square(Y, _, _), . .L) :–
        printlist(L),
        prin(Y).
printlist([]).
```

(The built-in predicate *prin* writes the value of *Y* but, unlike *print*, does not start a new line.)

Here we scan through the list of solution *squares* and write out only the column number of each square. But note that the column numbers are not written in the order in which they appear in the solution list. Because we make the recursive call to *printlist* before writing, we actually write out the column numbers in the order in which they were inserted. That is, the column number for the first row is written out first.

This abbreviated list of column numbers is an improvement, but a much clearer form of output is produced by the final alteration to our program:

```
print_board(N, [square(X, _, _), . .L]) :–
        nl,
        (Y is X – 1),
        write_dots(Y),
        prin('Q'),
        (W is N – X),
        write_dots(W),
        print_board(N, L).
print_board(N, []) :– nl.
```

```
write_dots(X) :-
        X > 0,
        prin('.'),
        (Y is X - 1),
        write_dots(Y).
write_dots(0).
```

The only thing that needs explanation here is the built-in predicate *nl*. This is a command to start a new line on the current output stream.

Using the University of New South Wales implementation on a VAX 11/780, it takes PROLOG a little over one minute to respond to the directive

```
:- queens(8, X), print_board(8, X), fail.
```

by producing 92 solutions in this form

```
. . . Q . . . .
. Q . . . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .
. . . . . . . Q
. . . . Q . . .
Q . . . . . . .

. . . . Q . . .
. Q . . . . . .
. . . Q . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . . . Q
. . . . . Q . .
Q . . . . . . .

.
.
.
```

The time taken for execution of the program is about four times that taken by a PASCAL program (based on Dijkstra's algorithm). Apart from questions of efficiency of implementation of the two languages, this time difference is due largely to the fact that testing for the safety of a square in PASCAL is less time-consuming. One merely has to look at a single array element for each diagonal. However, in terms of ease of programming, the PROLOG program is unquestionably more straightforward.

5. BACKTRACKING AND CONTROL

On a number of occasions in the preceding sections, we have mentioned that PROLOG's automatic backtracking leads to the generation of multiple solutions. Before proceeding to our final sample program, we will examine this process in a little more detail and introduce a feature of PROLOG which allows the programmer to inhibit undesirable or unnecessary backtracking.

Consider the procedure introduced in section 4 to choose an element from a list:

```
choose(X, [X . .Y], Y).
choose(X, [H, . .T1], [H, . .T2]) :-
        choose(X, T1, T2).
```

If we assume that we are attempting to solve the 4-queens problem then, the first time we call *choose*, it is with the goal

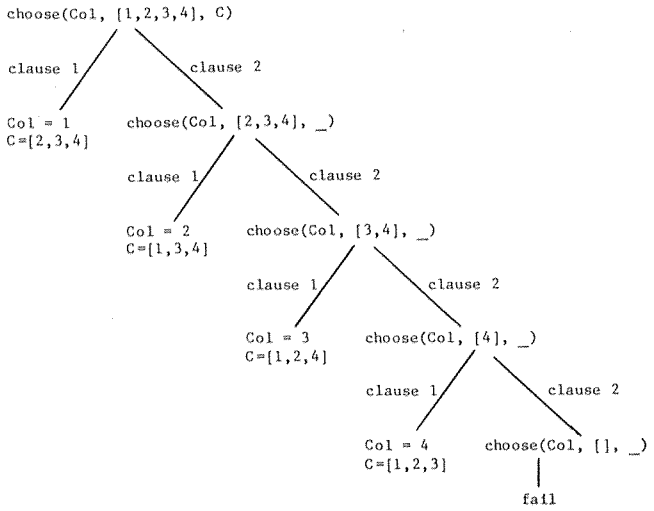
```
choose(Col, [1,2,3,4], C).
```

This goal (let's call it *G1*) will match the head of both clauses in the procedure but PROLOG automatically selects the first one (simply because it appears first). As a result, *Col* is unified with the integer, 1, and *C* with the list, [2,3,4]. If a failure occurs in a later goal (e.g. *safe*) then PROLOG backtracks to the last point where it made a choice — which was the decision to match *G1* with the first clause of the procedure. It "forgets" the values given to *Col* and *C* as a result of that match and finds an alternative match, namely, clause two.

Now the second clause attempts to satisfy the goal *(G2)*

```
choose(Col, [2,3,4], T2)
```

and again has two alternatives. On the first backtrack *G2* will match the head of clause one so that *Col* will become 2 and *C* becomes [1,3,4]. When yet another failure causes backtracking to *G2*, the second clause will be matched giving rise to two more alternatives, and so on. The sequence of events is most clearly illustrated by the following tree. At each node, the first choice is to take the left branch but backtracking forces us down the right branch until no further matches are possible.



When we reach the end of the right hand branch, the goal *choose* will fail completely and we either backtrack further to a preceding goal or, if there are no such goals, fail the original directive.

Although this is a relatively simple example, it illustrates the mechanism by which multiple solutions are generated. Another aspect of the language is also illustrated here. While it is true (as we stated in section 2 when describing the basic structure of a PROLOG program) that each individual clause has a self-contained meaning which can be understood independently of other parts of the program, the sequencing of clauses can affect the actual execution of the program. For example, if we reverse the order of the clauses within the procedure *choose* then the first element chosen from the list [1,2,3,4] will be 4, the next 3, and so on.

The sequencing of clauses or of goals within a clause is, thus, a piece of control information which we have added to the logical relationships in our programs. The one other form of control which PROLOG allows is the so-called *cut* operator, "!", which is used to control backtracking in a program. This operator is inserted in a program just like any other goal but is not regarded as part of the logic of the program. Its effect is illustrated by the following example.

Suppose we have a directive of the form

```
:- goal1, goal2, goal3.
```

where the procedure for *goal2* has two clauses

```
goal2 :- P1, !, P2.
goal2 :- Q1, Q2.
```

Now let us assume that *goal2* in the directive matches the head of the first clause and that goal *P1* in the body of that clause is satisfied. When it is first encountered, the *cut* also succeeds so we move on to goal *P2*. At the same time, the *cut* discards all alternatives for *P1* and for *goal2* itself. So if either *P2* or some subsequent goal fail, causing backtracking to return to the *cut* then, because there are no alternatives available, the parent goal (*goal2* in the directive) will fail. So backtracking returns to *goal1* to attempt to re-satisfy that.

Recall that, in the absence of the *cut*, if *P2* had failed then we would have attempted to re-satisfy *P1* and, if that had also failed, we would have tried the alternative clause for *goal2*. The effect of the *cut* is to commit the system to all choices made between the invoking of the parent goal and the *cut* itself. So if the result of the first passage from the parent goal to the *cut* is not acceptable, then no others are sought.

Because it limits the degree of backtracking that takes place, the *cut* is used in some applications to improve the efficiency of a program. An example of how it might be used in this way is the following.

Consider the predicate *insert(X, L1, L2)* in section 3. For any given element, *X*, and list *L1*, only one of the clauses in that procedure can ever apply. To avoid the waste of time which might occur if backtracking ever led PROLOG to attempt to re-satisfy that goal by attempting to match inappropriate clauses, we could rewrite the procedure as

```
insert(X, [X,..T], [X,..T]) :- !.
insert(X, [H,..T1], [H,..T2]) :- X > H, !,
                                 insert(X, T1, T2).
insert(X, L1, [X,..L1]).
```

Notice that with *cuts* included in each of the first two clauses, PROLOG can only reach the third clause if either *X* is smaller than the head of *L1* or *L1* is the empty list. The third and fourth clauses of the original procedure can therefore be combined to give this simpler form without the need for *L1* to be explicitly exhibited.

Now let us look at our final sample application of PROLOG.

6. PROLOG IN COMPUTER-AIDED DESIGN

In this section, we will demonstrate the use of PROLOG in a very-small-scale computer-aided design application. While we shall be tackling a very specific problem, our approach is closely related to programs written for such diverse applications as compiler writing (Warren, 1980), the coding of grammar rules (Clocksin and Mellish, 1981) and symbolic algebra (Bergman and Kanoui, 1973). So the reader who is not interested in the specific model to be programmed should, nevertheless, find the techniques used of interest.

We wish to write a program which, given an arbitrary logical expression such as

```
~ a^ b v c^ ~d => e^ f                    (E)
```

[where the symbols represent the operations *not(~)*, *and(^)*, *or*(v) and *implies* (=>)], will output a sequence of instructions for constructing an equivalent electrical circuit using only *nand* gates, where *nand(x, y)* is equivalent to *~(x^y)*.

The problem falls into two parts. The first is to use the various rules of logic to *translate* the expression into one involving only the *nand* operation and the second is to *encode* this expression into a sequence of instructions suitable for use in, say, an automatic wire-wrap machine. We will take as our model the output of the "Wirit" programs of Hayes and Carrington (Vickers, 1981).

A simple expression such as

```
a v b
```

is translated into

```
nand(nand(a, a), nand(b, b))
```

and encoded as

```
gate(1)  7400
         in(1)   a
         in(2)   a

gate(2)  7400
         in(1)   b
         in(2)   b

gate(3)  7400
         in(1)   gate(1)/out
         in(2)   gate(2)/out

final    gate(3)/out
```

The circuit consists of three gates. These are contained in *7400* chips which are *nand* gates. For each gate, two inputs must be specified. Thus *gate(3)* receives inputs from the outputs of *gate(1)* and *gate(2)*. In addition, the final result is specified as the output from the third gate. (Of course in this example, because we assume that only two-input *nand* gates are available, it is not really necessary to specify the gate type in the output. However the principles used in this example can be extended to more general design problems where a variety of devices may be available.)

The translation phase is carried out by the procedures below:

```
:- op(700, xfx, =>).
:- op(600, yfx, v).
:- op(500, yfx, ^).
:- op(400, fx, ~).

translate(X, Y) :-
        trans(X, A),
        terminate(X, A, Y).

trans(~A, nand(B, B)) :- !, translate(A, B).
trans(nand(nand(B, B), nand(B, B)), B):- !.
trans(A => B, nand(C, nand(D, D))) :- !,
        translate(A, C),
        translate(B, D).
trans(A ^ B, nand(nand(C, D), nand(C, D))) :- !,
        translate(A, C),
        translate(B, D).
trans(A v B, nand(nand(C, C), nand(D, D))) :- !,
        translate(A, C),
        translate(B, D).
trans(nand(A, B), nand(C, D)) :- !,
        translate(A, C),
        translate(B, D).
trans(A, A).

terminate(X, X, X) :- !.
terminate(X, Y, Z) :- translate(Y, Z).
```

The first four clauses in the program are directives defining the atoms '=>', v, ^ and ~ to be operators. The arguments associated with the functor *op* specify precedence and associativity rules obeyed by the operators but we will not go into this here. Suffice it to say that the logical operators satisfy all the usual rules of logic. (A detailed discussion of operator definitions can be found in Clocksin and Mellish, 1981).

We actually translate a logical expression *X* into an equivalent form, *Y*, involving only *nand* operations by satisfying the goal *translate(X, Y)*. The procedure *translate* first calls on procedure *trans* to transform *X* into an equivalent

expression *A* and then calls on procedure *terminate* to check whether *A* can be simplified by further translation. Let's look at these two steps in more detail.

Procedure *trans* embodies all the logical rules required to translate an arbitrary expression into one involving only *nand* operations. For example, the first clause says that *not A* is equivalent to *nand(B, B)* where *B* is the translation of *A*. The second says that *not(not(B))* is equivalent to *B*. The remaining clauses represent similar rules for the *implies*, *and*, *or* and *nand* operations. The final clause ensures that expressions which do not involve any of the logical operators, such as the individual atoms in a logical expression are translated into themselves.

When the goal *trans(X, A)* in the main procedure is satisfied, expression *A* will be logically equivalent to *X* and will involve only *nand* operations. However, *A* may not be in its simplest form. For example, if we ask the question

```
? trans(a v a, X).
```

the response will be

```
X = nand(nand(a, a), nand(a, a))
```

whereas we know that this can be simplified considerably. In general, an expression *A* is only in its simplest form if the goal *trans(A, Y)* is satisfied with *Y* = *A*.

In our program, the procedure *terminate* is used to check whether a transformation has gone as far as possible. The goal *terminate(X, A, Y)* in procedure *translate* will match the head of the first clause in *terminate* if *A* = *X*. In that case, we know that the translation is completed so *Y* is instantiated to the value of *A* and this value is returned as our final answer. On the other hand if *A* is not equal to *X* then our goal matches the head of the second *terminate* clause. The body of this clause calls *translate* again, and this iterative procedure continues until no further translation is possible. (Note the *cut* in the body of the first clause. This indicates that if a failure in a later goal returns us to the procedure *terminate* then there is no point in attempting to simplify the translation again using the second clause as we have already found the simplest form.)

At the end of the translation phase, the logical expression, *a* v *b* will have been transformed into the structure *nand(nand(a, a), nand(b, b))* as we required. The task of encoding this translation into a sequence of instructions for constructing the circuit is performed by the following procedures.

```
wirit(nand(A, B), gate(M)/out, N, M) :- !,
        wirit(A, OutA, N, NA),
        wirit(B, OutB, NA, NB),
        M is NB + 1,
        print_nand(M, OutA, OutB).
wirit(X, X, N, N).


print_nand(GateNumber, Input1, Input2) :-
        print(gate(GateNumber), '7400'),
        print(' in(1)  ', Input 1),
        print(' in(2)  ', Input 2).
```

The first argument of *wirit* is the structure produced by *translate*. The second argument represents the output of the gate. The third and fourth arguments are used to number the gates. *N* will be the number used to start the numbering sequence and *M* will be the last number in the sequence.

To follow the workings of *wirit*, we reproduce below a trace of execution of the goal:

```
wirit(nand(nand(a, a), nand(b, b), Final, 0, M)
    wirit(nand(nand(a, a), nand(b, b)), gate(M)/out, 0, M)
        wirit(nand(a, a), gate(M)/out, 0, M)
            wirit(a, a, 0, 0)
            wirit(a, a, 0, 0)
            M is 0 + 1
            print_nand(1, a, a)
        wirit(nand(b, b), gate(M)/out, 1, M)
            wirit(b, b, 1, 1)
            wirit(b, b, 1, 1)
            M is 1 + 1
            print_nand(2, b, b)
        M is 2 + 1
        print_nand(3, gate(1)/out, gate(2)/out)
```

Each line in the trace represents a call to a procedure. Notice that the first clause of *wirit* calls itself recursively twice, one call for each input to the *nand* gate. *M* is used to number the gates as they are created. Thus each time a new gate is printed, *M* is incremented. The final call is to the procedure *print_nand* which accepts as its input the number of the *nand* gate and the two inputs. This information is printed, resulting in the output shown earlier.

If we define a predicate *design* by

```
design(x) :-
        translate(X, Y),
        wirit(Y, Final, 0, _), nl,
        print('final   ', Final).
```

then the directive

```
:- design(a v b).
```

will result in the description given earlier in this section. To learn how to construct a circuit representing our "arbitrary logical expression", *E*, we simply ask

```
:- design(~a^b v c^~d => e^f).
```

and receive the response

```
gate(1)  7400
        in(1)   a
        in(2)   a

gate(2)  7400
        in(1)   gate(1) / out
        in(2)   b

gate(3)  7400
        in(1)   d
        in(2)   d

gate(4)  7400
        in(1)   c
        in(2)   gate(3) / out

gate(5)  7400
        in(1)   gate(2) / out
        in(2)   gate(4) / out

gate(6)  7400
        in(1)   gate(5) / out
        in(2)   gate(6) / out

final   gate(7) / out
```

To be honest now, we really should point out that while our program will always produce a set of instructions like this, it may not necessarily produce the best set. For example, the *exclusive-or* operation can be represented by either of the equivalent expressions

```
(a^~b) v (~a^b)
```

or

```
(a v b) ^~(a^b)
```

If we *design* each of these expressions, we find that while the first leads to the simplest five-*nand* configuration, the second form requires six *nand* operations. This is a consequence of the fact that our translation program is really very simple and we have not built in such information as

the distributive laws for logical operations. However, our aim was to illustrate what can be done with a simple program rather than to produce an optimum design.

7. CONCLUSION

With the help of a number of specific examples, we have illustrated many of the unique features of the PROLOG language. However, in such a brief introduction, there are some important features which we have not been able to mention at all such as the ability to dynamically *assert* and *retract* clauses within a PROLOG program, the variety of input-output procedures available (apart from *prin* and *print*) and built-in predicates which allow the programmer access to the component terms of a clause or general structure. These built-in features are not essential to an understanding of the way PROLOG works but enhance the power of the language in practical applications.

The basic concepts discussed in this paper are central to any PROLOG implementation but no "standard" PROLOG has yet been established. Specific details such as the names of built-in predicates or whether a question mark is placed at the beginning or end of a question may, therefore, vary from one implementation to another. The format in this paper conforms with the UNSW-PROLOG implementation (Version 4) which is described in a companion paper (Sammut and Sammut, 1983) and grew out of DEC-10 PROLOG (Warren, 1977). Two of the other major systems currently in use are IC-PROLOG (Clark and McCabe, 1979) and Micro-PROLOG (McCabe, 1981) which is designed for micro-computers based on the Z80 processor.

Information on available implementation and on new applications of PROLOG can be found in the *Logic Programming Newsletter* (published by Universidade Nova de Lisboa, Portugal).

8. REFERENCES

BERGMAN, M. and KANOUI, H. (1973): *Application of mechanical theorem proving to symbolic calculus*, Group d'Intelligence Artificielle, UER Luminy, Universite d'Aix-Marseille.

CLARK, K.L. and McCABE, F.G. (1979): Control facilities of IC-PROLOG. In *Expert Systems in the Microelectronic Age* (Ed. D. Michie), Edinburgh University Press.

CLOCKSIN, W.F. and MELLISH, C.S. (1981): *Programming in PROLOG*, Springer-Verlag, Berlin.

COLMERAUER, A. (1982): An interesting subset of natural language. In *Logic Programming* (Eds. K.L. Clark and S.-A. Tarnlund), Academic Press, London, pp. 45-66.

DIJKSTRA, E.W. (1972): Notes on structured programming. In *Structured Programming* (O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare), Academic Press, London, pp. 72-82.

van EMDEN, M.H. (1977): Programming with resolution logic, *Machine Intelligence 8* (Eds. Elcock, E.W. and Michie, D.), Ellis Horwood, Chichester, pp. 266-299.

van EMDEN, M.H. (1978): Computation and deductive information retrieval. In *Formal Description of Programming Concepts* (Ed. E. Neuhold), North-Holland, Amsterdam, pp. 421-440.

KOWALSKI, R.A. (1974): Predicate logic as programming language, *Proc. IFIP-74*, North-Holland, Amsterdam, pp. 569-574.

LLOYD, J.W. (1983): An introduction to deductive database systems, *Austral. Comput. J.*, **15**, pp. 52-57.

McCABE, F.G. (1981): *Micro-PROLOG Programmer's Reference Manual*, Logic Programming Associates Ltd., London.

MARKUSZ, Z. (1977): How to design variants of flats using programming language PROLOG based on mathematical logic, *Proc. IFIP-77*, North-Holland, Amsterdam, pp. 885-890.

SAMMUT, C.A. and SAMMUT, R.A. (1983): The implementation of UNSW-PROLOG, *Austral. Comput. J.*, **15**, pp. 58-64.

SANTANE-TOTH, E and SZEREDI, P. (1982): PROLOG applications in Hungary. In *Logic Programming* (Eds. K.L. Clark and S.-A. Tarnlund), Academic Press, London, pp. 19-31.

SERGOT, M. (1982): Prospects for representing the law as logic program. In *Logic Programming* (Eds. K.L. Clark and S.-A. Tarnlund), Academic Press, London, pp. 33-42.

VICKERS, T. (1981): *Condor – A Connection Descriptor and Query System*, Computer Science Honours Thesis, University of New South Wales.

WARREN, D.H.D. (1976): Generating conditional plans and programs, *Proc. AISB Conf.*, Edinburgh, pp. 344-354.

WARREN, D.H.D. (1980): Logic programming and compiler writing, *Software – Practice and Experience* **10**, pp. 97-125.

WARREN, D.H.D. (1982): A view of the fifth generation and its impact, *AI Magazine*, Fall 1982, pp. 34-39.

BIOGRAPHICAL NOTES

Rowland Sammut received his BSc degree in Applied Mathematics from the Univesity of New South Wales in 1973 and his PhD degree from the Australian National University in 1976. During 1976-77, he was a Research Fellow with the Optical Communications group at the University of Southampton.

He returned to the ANU in 1977 as a Radio Research Board Fellow and was later awarded a Queen Elizabeth II Fellowship to work on Optical Communications in the Applied Mathematics Department of the Research School of Physical Sciences. Since July 1981, he has been a lecturer in the School of Electrical Engineering and Computer Science of the University of NSW. Dr Sammut has approximately 50 publications in various areas of photoreceptor optics and optical waveguide theory.

Claude Sammut received his BSc in 1978 and his PhD in 1982, both from the Department of Computer Science in the University of New South Wales. His work in concept learning and the representation of concepts in predicate logic led to the development of the UNSW-Prolog system. He is currently a Post-Doctoral Fellow at Saint Joseph's University in Philadelphia where he is continuing his research in concept learning.

An Introduction to Deductive Database Systems

By J. W. Lloyd*

This paper gives a tutorial introduction to deductive database systems. Such systems have developed largely from the combined application of the ideas of logic programming and relational databases. The elegant theoretical framework for deductive database systems is provided by first order logic. Logic is used as a uniform language for data, programs, queries, views and integrity constraints. It is stressed that it is possible to build practical and efficient database systems using these ideas.

Keywords and phrases: Database management, query languages, question answering systems, theorem proving, logic programming, deductive database, PROLOG.
CR Categories: H.2.3, H.3.4, I.2.3.

1. INTRODUCTION

Deductive database systems have developed largely from the combined application of the ideas of logic programming and relational database systems. They are called "deductive" because they are able to make deductions from known facts and rules when answering user queries. (Deductive databases have also been called logic databases [Gallaire and Minker, 1978], deductive relational databases [Minker, 1982] and virtual relational databases [Debenham and McGrath, 1983].) In other words, they have a limited reasoning ability. This deductive capability has come from research in artificial intelligence, where considerable success has been achieved in building systems capable of intelligent activity, such as understanding natural language and problem solving.

Deductive database theory subsumes the more standard relational database theory. A relational database consists of a collection of facts. Deductive databases contain not only facts, but also general rules. Thus a relational database is a special kind of deductive database. First order logic can be used to express naturally both facts and rules. Indeed, we shall see that logic can be used to express not only data, but also programs, queries, views and integrity constraints.

The other inspiration for deductive database systems comes from logic programming. This is based on the premise that first order logic can be used as a programming language. There has been a considerable amount of interest in this area in recent years. So far, the main result of research into logic programming has been the programming language PROLOG (PROgramming in LOGic). PROLOG is a simple but very powerful language with applications in many areas, including artificial intelligence and database systems.

Using PROLOG as an implementation language, it is easy to provide user-friendly query languages, such as SQL and QBE (Date, 1981), as a user interface to a deductive database system. A number of deductive database systems also provide (a limited subset of) English, as well as other natural languages, as a query language. PROLOG is very convenient for implementing query languages because it provides a grammar rule notation, which can be used to implement parsers. In fact, PROLOG was originally developed as a tool for building natural language understanding programs.

In later sections, we briefly discuss logic programming, describe the foundations of deductive databases, discuss some implementation details and outline some systems which have been implemented in recent years.

2. LOGIC PROGRAMMING

This section contains a brief introduction to logic programming. Much more complete discussions are given by Sammut and Sammut (1983b), Clocksin and Mellish (1981), Coelho *et al.* (1980) and Kowalski (1979). A collection of advanced papers is contained in Clark and Tarnlund (1982). Our purpose here is merely to highlight the aspects of logic programming which are important for deductive database systems.

The most common logic programming systems available today are PROLOG interpreters. Consequently, we confine the discussion to this setting. We begin with a very simple example of a PROLOG program:

```
grandparent(x,y) :- parent(x,z),parent(z,y)

parent(x,y) :- mother(x,y)
parent(x,y) :- father(x,y)

ancestor(x,y) :- parent(z,y),ancestor(x,z)
ancestor(x,y) :- parent(x,y)

father(fred,mary)
father(george,james)
father(john,fred)

mother(sue,mary)
mother(jane,sue)
mother(liz,fred)
mother(sue,james)
```

The syntax for PROLOG programs is not by any means standardized. However, current systems have syntax close to that above. The program consists of a collection of *(program) clauses*, such as ancestor(x,y) :- parent(x,y). A construct such as ancestor(x,y) is called an *atom*. Each atom has a *predicate* symbol followed by some arguments. Thus grandparent, parent, ancestor, . . . are predicates. The arguments can either be *variables* (such as x,y,z, . . .) or *constants* (such as fred, mary, john, . . .). The programmer is free to name the various predicates, constants and variables in any way desired (subject to the restrictions of syntax, of course).

*Department of Computer Science, University of Melbourne, Parkville, Victoria 3052. *Manuscript received September 1981; revised April, 1983.*

Each clause is shorthand for a first order logic formula. The convention is that all variables which appear in the clause are universally quantified at the front. Thus the so-called *declarative* meaning of

ancestor(x,y) :– parent(x,y)

is

for all x and y, if x is a parent of y, then x is an ancestor of y.

Similarly, the clause

ancestor(x,y) :– parent(z,y),ancestor(x,z)

means

for all x,y and z, if z is a parent of y and x is an ancestor of z, then x is an ancestor of y.

Thus, the :– is ordinary logical implication (usually written in logic as ←) and the comma between atoms to the right of the :– stands for logical conjunction (and).

The single atom on the left of the :– is called the *head* of the clause, while the atoms on the right of the :– are together called the *body* of the clause. Notice that some clauses have an empty body. Such clauses are unconditional and are simply *facts*. Thus father(fred,mary) states that fred is a father of mary. The clauses with non-empty body are conditional and are called *rules*. They state that something holds *if* something else holds. A PROLOG program is simply a collection of facts and rules. We call the collection of clauses which have the same predicate p, say, in the head, the *procedure about* p. Thus we can also consider a PROLOG program to be a collection of procedures about certain predicates. In our example, there are five procedures.

To run a PROLOG program, a *query* (also called a *goal*) is given to the interpreter. For example, to find out who is the father of fred, the query

:– father(x,fred)

is used. A query is a clause with a body, but with no head. The declarative reading of the above query is

find an x such that x is the father of fred.

The interpreter would respond with the *answer* x=john.

Many queries have more than one answer. For example, the query :– mother(sue,x) asks for the children of sue. The answers are x=mary and x=james. The answers are returned one at a time by the interpreter after suitable prompts from the user. However, if desired, it is a simple matter to modify the above query so that all answers are returned together in a set.

The arguments in a query which are constants are used for *input* to the program. Arguments which are variables are used for *output*. One distinctive feature of logic programming is that whether an argument is used for input or output is not predefined. The user may specify any input-output pattern desired. For example, the query :– parent(fred,mary) simply requires a yes/no answer, as there are no variables and hence no output. The answer is yes, because fred is the father of mary and if a person is a father, by the second rule for parent, that person is also a parent. The query :– parent(x,y) asks for all pairs x,y such that x is a parent of y. There are seven answers to this query.

Sometimes we are not interested in the values of all the variables which appear in a query. Suppose we want to know all those persons who are fathers. For this we can use the query

:– x:father(x,y).

The "x:" means "only return the value for x; the corresponding y value is not of interest". This query would return the answers x=fred, x=george and x=john. More generally, we can put a list of variables separated by commas before the colon. Queries may also contain more than one atom. For example,

:– mother(sue,x),father(george,x)

asks for persons whose mother is sue and whose father is george.

How does the PROLOG interpreter compute an answer to a query? Consider the query :– grandparent(john,x). A (correct) answer to this query has the property that grandparent(john,x), with x replaced by the value in the answer, can be *deduced* (in the ordinary logical sense) from the program. Now x=mary is an answer to the query. This means that grandparent(john,mary) can be deduced from the program. For this particular query, the deduction involves the use of the rule for grandparent, the rule parent(x,y) :– father(x,y) and the facts father(john,fred) and father(fred,mary).

The details of how this deduction is performed are beyond the scope of this paper. The reader can find discussions in Sammut and Sammut (1983a), Kowalski (1979) and Lloyd (1982). However, we can say that the theoretical foundations for logic programming have come directly from work in automatic theorem proving in artificial intelligence in the late 1960's. In fact, during the answering of a query, the interpreter is actually proving a theorem. The statement to be proved is the query and the axioms used to prove the statement are the program clauses. Each step in the query answering process is a step in the proof of the query. In theorem proving terminology, each step is a deduction using the *resolution* inference rule. The major part of resolution is a pattern matching process called *unification*. During unification, variables become bound to constants and other variables. This is how an answer gets constructed.

The *procedural* meaning of a PROLOG program is provided by the behaviour of the interpreter. However, one of the major features of logic programming is that (to a large extent) it is not necessary to know the behaviour of the interpreter! In fact, the ideal of logic programming is to be able to simply write down declaratively correct clauses about what is to be computed and leave the actual problem of computation entirely to the interpreter. Unfortunately, this ideal has not yet been achieved with current systems.

There are numerous PROLOG systems in use all over the world. Some well-known ones are DECSYSTEM-10 PROLOG (Pereira *et al.*, 1979), UNIX PROLOG (Clocksin and Mellish, 1980) and micro-PROLOG (McCabe, 1981), which runs on Z80 based microcomputers. At the University of Melbourne, L. Naish is developing a new system called MU-PROLOG (Naish, 1982a and 1982b). It is written in C and intended for a UNIX environment. MU-PROLOG is compatible with DECSYSTEM-10 and UNIX PROLOG, but has a number of novel features. These features give the interpreter a more sophisticated control of the deduction process and are a step towards the ideal of logic programming – purely declarative programming.

PROLOG has been demonstrated to be easy to learn, write and modify. PROLOG is also efficient. Interpreted PROLOG is about as fast as interpreted LISP and compiled PROLOG is comparable in speed with more conventional languages.

An outstanding recent development, which promises to make logic programming an area of major importance, is Japan's fifth generation computer system project (Treleaven and Gouveia Lima, 1982). The aim of the

project is to develop computer systems for the 1990's. This will involve bringing together four currently separate areas of research: expert systems, very high level programming languages, distributed computing and VLSI technology. It is intended that the very high level languages be based on logic and that computer architecture be designed to directly support such languages.

The reader should appreciate that we have given only the briefest glimpse of logic programming in this section and a fuller understanding of the field would require a close study of the references. With this understanding, let us now discuss deductive database systems.

3. DEDUCTIVE DATABASE SYSTEMS

A *deductive* database is, from a conceptual point of view, simply a PROLOG program. Thus it consists of a collection of facts and rules. However, there is a practical difference between PROLOG programs and deductive databases. A program generally consists mostly of rules, together with some facts. On the other hand, a deductive database would normally consist of only a small number of rules, but thousands, even hundreds of thousands, of facts. This implies that it is impossible to "load" a deductive database by reading it all at once into main store. PROLOG programs are usually small enough to be kept in certain data structures in main store so that they can be easily accessed by the interpreter. A deductive database must be kept on disk and only those parts of it required to answer the current query are read into main store. Thus a deductive database system requires file structures similar to a relational database system so that the interpreter can quickly access any fact or rule it requires. We will return to this problem in Section 4.

The first task is to show that a relational database is a special kind of deductive database. To illustrate this point, we use the supplier-part relational database from Date (1981, p. 92). This database has three relations: a relation s of suppliers, a relation p of parts and a relation sp, which contains both supplier and part information. More precisely, the relations have the following form (where the domain names are self-explanatory):

```
s(sno,sname,status,city)
p(pno,pname,colour,weight,city)
sp(sno,pno,qty)
```

Relations are usually represented by tables containing tuples. However, we prefer to write the tuples using the syntax we introduced earlier for PROLOG programs. Here is the supplier-part database:

```
s(s1,smith,20,london)
s(s2,jones,10,paris)
s(s3,blake,30,paris)
s(s4,clark,20,london)
s(s5,adams,30,athens)

p(p1,nut,red,12,london)
p(p2,bolt,green,17,paris)
p(p3,screw,blue,17,rome)
p(p4,screw,red,14,london)
p(p5,cam,blue,12,paris)
p(p6,cog,red,19,london)

sp(s1,p1,300)
sp(s1,p2,200)
sp(s1,p3,400)
sp(s1,p4,200)
sp(s1,p5,100)
sp(s1,p6,100)
sp(s2,p1,300)
sp(s2,p2,400)
sp(s3,p2,200)
sp(s4,p2,200)
sp(s4,p4,300)
sp(s4,p5,400)
```

The tuples of the relations have been written like facts in a PROLOG program. In fact, apart from minor syntactic differences, tuples and facts *are* the same. (The reason is that in first order logic, relations and predicates can be identified in a very trivial way).

What this example illustrates is that a *relational database is just a special type of PROLOG program*, one which consists solely of facts. Thus a beautiful idea emerges. Why not generalize relational databases by allowing them to contain not just facts, but also rules? We thus arrive at the idea of a deductive database as a result of the combination of relational database theory and logic programming. Furthermore, logic now provides the theoretical framework for database systems. The advantage of logic over other possible contenders is that logic has been intensely studied and used for many decades, it is a powerful data modelling tool, its semantics is very well understood and, finally, a logic programming language can be used to easily implement a variety of query languages and other facilities. It is interesting to note that the central importance of logic to database systems has become widely appreciated in the literature in the last couple of years.

The supplier-part database could be made into a deductive database by adding rules. For example, we might add

```
london-s(x,y,z) :- s(x,y,z,london)
major-s(x) :- sp(x,p1,z), z >= 300
```

These rules implicitly define two new relations

```
london-s(sno,sname,status)
major-s(sno)
```

Sometimes such relations are called *virtual* relations because when a tuple from the relation is wanted, it must be *computed* rather than *retrieved*. The first rule states that a london-supplier is any supplier based in London. The second rule states that a supplier is a major-supplier if it supplies more than 300 of part number p1.

As a further example, consider a database which describes the activities of a university department. Included in the database could be facts such as

```
teaches(jwl,303)
teaches(jwl,486)
size(303,50)
day(303,monday)
```

The facts state that jwl teaches courses 303 and 486, the enrollment in 303 is 50 students and 303 is held on Mondays. These are the sort of things a relational database would contain. But we can also include rules such as

```
occupied(x,y) :- teaches(x,y)
occupied(x,y) :- attends(x,y)
attends(x,141) :- year(x,1),major(x,compsci)
```

The first two rules could be used for timetabling purposes. They state that a person x is occupied with a course y if x teaches y or x is a student attending y. The third rule states that every student who is in first year and is majoring in computer science must attend course 141.

Notice the power of this last rule It tells us something about *all* first year computer science majors. Using it, we can *deduce* that all such students attend 141. Thus we see that deductive databases offer the expressibility of logic for data modelling, while allowing the possibility of considerable space saving by having large numbers of tuples collapse into a general rule.

As in any database system, we will be interested in deleting, adding, modifying and retrieving information. It is well-known that update anomalies can occur in relational databases. Similar problems may occur in deductive databases and they can be avoided by the usual techniques of storing facts in normal form (say, fourth normal form [Date, 1981]). Of more interest is the retrieval of information. Deductive database systems provide what we call the *standard* query language. This is just the specification of a query as in PROLOG programming. Thus to find out the names of all lecturers who lecture third year units, we

might ask

```
:– x:lectures(x,y),level(y,3).
```

The standard query language is probably too primitive for casual database users. What is required are query languages using subsets of English or, at least, structured English. As we have indicated, PROLOG is very suitable for implementing query languages such as SQL, QBE (Neves *et al.*, 1983) or even natural language (Dahl, 1982), (Warren and Pereira, 1981).

A database facility for MU-PROLOG is being developed at the University of Melbourne. The facility uses a sophisticated file structure (see section 4) and will ultimately support a number of query languages. Currently, SQL is supported, as well as the standard query language. SQL, as described in (Date, 1981, ch. 7), has been implemented in (MU-)PROLOG by W. Bray. The overall process of answering an SQL query is as follows. The query is parsed, a PROLOG query is generated, some optimization is done (see Section 4) and then the optimized PROLOG query is run.

To give an example, the SQL query (Date, 1981, p. 121)

```
select sname
from s,sp
where s.sno = sp.sno
and sp.pno = 'p2'
```

is translated into (approximately) the PROLOG query

```
:– y:sp(x,p2,u),s(x,y,z,w)
```

This query asks for the names of suppliers which supply part p2. More complex SQL queries lead to PROLOG queries which have calls to "system" procedures such as count, sum, . . . (Date, 1981, p. 132), as well as calls to the database procedures. Our experience and that of Neves *et al.* (1983) shows that the task of implementing user-friendly query languages and other facilities via PROLOG is quite straightforward.

As well as asking queries of a database system, we will also want to do some processing of the data from time to time. For example, we will perhaps want reports produced giving summaries of an inventory database. As in a conventional system, a program has to be written. In a deductive database system, a PROLOG program would need to be written to produce the report. This program is nothing more than a set of extra clauses added to the database and initiated by a certain query which calls its top level procedure. Thus PROLOG is the host programming language.

Integrity constraints can also be included in a deductive database. These constraints are just rules. Data being added to the database is checked to see whether it is consistent with the constraints. If it is inconsistent, the data is rejected. The issues here are similar to those for relational databases and so we do not discuss them further. (See, for example, Gallaire and Minker [1978] and Gallaire *et al.* [1981].)

It is interesting to compare deductive database systems with a state-of-the-art relational database system, such as System R, which is discussed in detail in Date (1981). System R embodies most of the relational ideas in a powerful, elegant and practical database system. To what extent then can deductive databases improve on System R? In respect of concurrent access, back-up and recovery, and security and integrity, the facilities in both System R and deductive database systems would be very similar. However, deductive database systems do offer important advantages over System R, which we discuss below.

The host languages in System R are PL/1 and COBOL. Access to the database is provided by embedded SQL statements in a host language. However, SQL is so different from PL/1 and COBOL that the resulting combination is rather unsatisfactory. Besides programmers need to know *both* SQL and one of the host languages. In a deductive database, programmers need to know only one language, PROLOG, which can be used to ask queries and also used as the host language. Being based on logic, PROLOG is a much more suitable host language for databases than procedural languages such as PL/1 or COBOL.

It is also advantageous to have a single mechanism, the PROLOG interpreter, which is used for answering queries and also for running host language programs. Furthermore, it is possible to give the interpreter sufficient sophistication to optimize queries so that they can be answered with the minimum number of disk accesses (see Section 4).

In a deductive database system, there is also the advantage of rules. Now System R (and some other systems) do provide a similar facility, called a *view* (Date, 1981, ch. 9). A view is a virtual (or derived) relation that does not really exist in its own right, but is derived from one or more underlying relations. Thus a view is a special case of a rule. One can add "views" to a deductive database simply by adding rules. Rules thus share all the advantages that views bring. However, rules are rather more useful and powerful than views. First, rules are at the heart of a deductive database rather than on the outside, as views are in a relational database. Rules are used in an important way in the data modelling stage of database creation and become an essential part of the data model itself. For example, they provide the flexibility of defining a relation partly by some facts and partly by some rules. Furthermore, rules can be recursive (see the ancestor procedure earlier). This is a very useful property, which is not shared by views.

In general, relational database systems compensate for their inability to directly express general laws about data by interfacing with conventional programming languages. In other words, a general law, which can be expressed by a single rule, say, in a deductive database system, will normally need to be expressed *procedurally* in a relational database system by a program written in a host language. The rule is advantageous in many respects. It is more explicit, more concise, easier to understand and easier to modify than the corresponding program.

In this section, we have described an approach to deductive database systems based on PROLOG. In fact, this is just one possible approach. A deductive database system can be based on any logic programming language and, indeed, on any theorem prover. We refer the reader to Gallaire and Minker (1978) and Gallaire *et al.* (1981) for a discussion of a number of alternative approaches. However, no matter how varied these are, they are all based on logic. Thus they all share the advantages described here.

4. CLAUSE INDEXING AND QUERY OPTIMIZATION

In this section, we discuss two implementation problems for deductive database systems – clause indexing and query optimization. Both of these have a substantial effect on the cost of answering a query.

Earlier we mentioned that most deductive databases have only a small number of rules, which can therefore be kept in the internal data structures of the interpreter.

However, there would normally be so many facts that they have to be kept on disk. We are thus faced with the problem of finding a suitable file structure so that the facts can be retrieved efficiently when they are required by the interpreter. This is generally called the *clause indexing* problem.

There are two distinct solutions to this problem. One is the "compiled" approach and the other is the "interpreted" approach (Chakravarthy *et al.*, 1982). In the compiled approach, the interpreter first uses the rules to generate a number of subsidiary queries, which can then be answered by looking at the facts. Thus no fact is accessed until the very last step of the query answering process. In other words, the process is one of pure computation followed by one of pure retrieval. However, in the interpreted approach, the accessing of facts and the computation are interleaved. Thus whenever the interpeter needs a fact to continue the computation, a disk access is potentially required (potentially, because the required page may already be in main store).

In addition, we can classify solutions to the clause indexing problem according to whether the accessing of the facts is done using a specially implemented file structure or whether the facts are accessed using a conventional database system. In the first approach, the implementors have to devise and implement a suitable file structure themselves. In the latter approach, the deductive part of the database is essentially a "front-end" to a conventional database and all problems of disk access are left to the conventional database (Debenham and McGrath, 1982).

Let us now examine the interpreted approach in a little more detail. During the answering of a query, the interpreter generates a number of partial-match queries. A *partial-match query* is the specification of the values of some of the secondary keys of a record. An *answer* to a partial-match query is a listing of all records in the file which have the specified values for the specified keys. For example, at a certain stage during the answering of a query, the interpreter may need to know the sp facts which are related to part p5. Thus the partial-match query, with the pattern sp(*,p5,*), is generated. (The *'s stand for unspecified arguments.) The answer to this partial-match query is the set of facts { sp(s1,p5,100), sp(s4,p5,400) } . Using a suitable answer from this set, the interpreter is then able to continue the query answering process. The point here is that the interpreter is only interested in sp facts with p5 in the second argument. Thus we require a file structure which retrieves this particular subset of sp facts with the *minimum number of disk accesses.* In particular, a linear search through the sp relation is totally unacceptable.

Thus the clause indexing problem is one of finding an efficient partial-match retrieval scheme. We would like the design to be able to accommodate dynamic files because there are likely to be many deletions and insertions in the database. The classical solution to this problem is the inverted file. However, inverted files have many disadvantages, particularly for dynamic files. Much more elegant solutions can be found using the idea of multi-key hashing. Each secondary key has its own hashing function and the values from each hashing function are combined in a natural way to get a hashed value for the entire record. The details of this are beyond the scope of this paper. We refer the interested reader to Lloyd (1981), Lloyd and Ramamohanarao (1982), Ramamohanarao and Lloyd (1982) and Ramamohanarao *et al.* (1983).

The MU-PROLOG database facility follows the interpreted approach and uses the file structure from Ramamohanarao *et al.* (1983) to implement clause indexing. This implementation is being carried out by J. Thom. Early results are very encouraging and indicate that it is possible to build very large databases using this approach.

Now let us turn to the problem of query optimization. Consider once again the PROLOG query

```
:- y:sp(x,p2,u),s(x,y,z,w).
```

When answering this query, the PROLOG interpreter must "answer" each of the "subqueries" :- sp(x,p2,u) and :- s(x,y,z,w). Let us suppose it first attacks :- sp(x,p2,u). Because the second argument has been specified, there are only a few pairs of values for x and u which make sp(x,p2,u) true. For each of these pairs, the interpreter then looks to see if it can make s(x,y,z,w), with the x replaced by its value, true. Once again, because the first argument has been specified, there are only a few facts in the procedure about s which have to be looked at.

However, suppose instead the interpreter attacks the subquery :- s(x,y,z,w) first. Because no argument is specified, *every* fact in the procedure about s is retrieved. This is a very inefficient way of answering the query. It is clear from the other part of the query that only a small subset of the s facts is relevant.

This example shows that the query optimization problem is essentially that of finding an appropriate order for solving subqueries so that the total cost of answering the query is minimized. The order in which a PROLOG interpreter answers the subqueries of a query is called its *computation rule.* Standard PROLOG systems have a fixed left to right computation rule. That is, they always answer the leftmost subquery first, then the second to left, etc. In such systems, the query optimization must be done before the query is given to the interpreter. However, MU-PROLOG has a more sophisticated computation rule, not yet fully implemented, which essentially allows it to reorder subqueries during the answering of the query. This is a big advantage because having a procedure in the database which contains a mixture of facts and rules can make the other approach impractical. For further details on the MU-PROLOG computation rule, we refer the reader to Naish (1982a).

5. SOME IMPLEMENTED SYSTEMS

In this section, we briefly describe some deductive database systems which have been implemented in recent years. More details can be found in the references.

The first system, called Chat-80, was implemented by Warren and Pereira (1981). Chat-80 can answer queries given in a limited subset of English. The database contains information about world geography. It is not a particularly large database and thus can be kept entirely in the main store. Hence problems of access to disk were avoided. To give an idea of the sort of queries Chat-80 can handle, we list some below.

Which is the largest African country? (Answer: Sudan)

How many countries does the Danube flow through? (Answer: 6)

Which is the ocean that borders African countries and that borders Asian countries? (Answer: Indian Ocean)

Which country bordering the Mediterranean borders a country that is bordered by a country whose population exceeds the population of India? (Answer: Turkey)

A system similar to Chat-80 is described in Dahl (1982). However, the query language is Spanish rather than English. The use of a deductive front-end to a conventional database is explored in Debenham and McGrath (1982). One of the databases from Debenham and McGrath (1982) is very large (about 250 megabytes) and shows that the deductive approach is practical even for very large databases.

There are many industrial applications of PROLOG and deductive databases in Hungary (Futo *et al.*, 1978), (Santane-Toth and Szeredi, 1982). For example, Santane-Toth and Szeredi (1982) lists 28 separate applications, which range from a system which generates COBOL programs to a deductive database containing information on pests and pesticides. Finally, a number of more experimental systems are described in Gallaire and Minker (1978) and Gallaire *et al.* (1981).

There are no deductive database systems available commercially yet, nor are there likely to be any for a number of years. However, we feel that the number of experimental systems already implemented, their range of application and the impetus supplied by the fifth generation project, strongly support the thesis that deductive database systems are the way of the future.

6. CONCLUSION

This paper has been rather concise in its coverage of what is now a substantial field. Some topics have only been discussed very briefly and other important topics (such as negative information) have been omitted entirely. As we have seen, deductive database systems have strong links with first order logic, logic programming, theorem proving from artificial intelligence, relational database theory and indexing techniques from the theory of file structures.

First order logic can be used as a foundation for practical deductive database systems. Logic serves as a single uniform language which can be used for giving data definitions, integrity constraints, views, queries and programs. In fact, the conventional distinction between data and procedure disappears. A logic programming language, such as PROLOG, is a very convenient language in which to implement user-friendly query languages and other facilities.

Hopefully, some readers will be sufficiently intrigued by the ideas presented here to want to delve more deeply. For further reading, we suggest starting with van Emden (1978), Gallaire (1981), Dahl (1982) or Minker (1982). The collection of papers in Gallaire and Minker (1978) and Gallaire *et al.* (1981) is invaluable. Finally, we mention Reiter (1981), which gives a theoretical account of the impact of logic on relational database systems.

7. REFERENCES

CHAKRAVARTHY, U.S., MINKER, J. and TRAN, C. (1982): Interfacing Predicate Logic Languages and Relational Databases, *Proc. of First International Logic Programming Conference*, Marseille, pp. 91-98.

CLARK, K.L. and TARNLUND, S.-A. (Eds.) (1982): *Logic Programming*, Academic Press.

CLOCKSIN, W.F. and MELLISH, C.S. (1980): *The UNIX PROLOG System*, Software Report 5, Department of Artificial Intelligence, University of Edinburgh.

CLOCKSIN, W.F. and MELLISH, C.S. (1981): *Programming in PROLOG*, Springer-Verlag, Berlin.

COELHO, H., COTTA, J.C. and PEREIRA, L.M. (1980): *How to Solve it with PROLOG*, Laboratorio Nacional de Engenharia Civil, Lisbon.

DAHL, V. (1982): On Database Systems Development through Logic, *ACM Trans. on Database Systems*, 7, pp. 102-123.

DATE, C.J. (1981): *An Introduction to Database Systems* (Third Edition), Addison-Wesley, Reading.

DEBENHAM, J.K. and McGRATH, G.M. (1982): The Description in Logic of Large Commercial Data Bases: A Methodology Put to the Test, *Austral. Comput. Sci. Comm.*, 5, Perth, pp. 12-21.

DEBENHAM, J.K. and McGRATH, G.M. (1983): LOFE: A Language for Virtual Relational Data Base, *Austral. Comput. J.*, 15, pp. 2-8.

van EMDEN, M.H. (1978): Computation and Deductive Information Retrieval, in *Formal Descriptions of Programming Concepts*, E.J. Neuhold (Ed.), North Holland.

FUTO, I., DARVAS, F. and SZEREDI, P. (1978): The Application of PROLOG to the Development of QA and DBM systems, in Gallaire and Minker (1978), pp. 347-376.

GALLAIRE, H. (1981): Impacts of Logic on Databases, *Proc. of 7th International Conference on Very Large Data Bases*, France, pp. 248-259.

GALLAIRE, H. and MINKER, J. (Eds.) (1978): *Logic and Data Bases*, Plenum Press, New York.

GALLAIRE, H., MINKER, J. and NICOLAS, J. (Eds.) (1981): *Advances in Data Base Theory, Vol. 1*, Plenum Press, New York.

KOWALSKI, R. (1979): *Logic for Problem Solving*, Elsevier North Holland, New York.

LLOYD, J.W. (1981): Implementing Clause Indexing in Deductive Database Systems, TR 81/4, Department of Computer Science, University of Melbourne.

LLOYD, J.W. (1982): Foundations of Logic Programming, TR 82/7, Department of Computer Science, University of Melbourne.

LLOYD, J.W. and RAMAMOHANARAO, K. (1982): Partial-Match Retrieval for Dynamic Files, *BIT*, 22, pp. 150-168.

McCABE, F.G. (1981): *Micro-PROLOG Programmer's Reference Manual*, Logic Programming Associates Ltd.

MINKER, J. (1982): On Deductive Relational Databases, *Fifth International Conference on Collective Phenomena.*

NAISH, L. (1982a): An Introduction to MU-PROLOG, TR 82/2, Department of Computer Science, University of Melbourne.

NAISH, L. (1982b): MU-PROLOG 2.5 Reference Manual, Department of Computer Science, University of Melbourne.

NEVES, J.C., BACKHOUSE, R.C., ANDERSON, S.O. and WILLIAMS, M.H. (1983): A PROLOG Implementation of Query-By-Example, *7th International Computing Symposium*, Germany.

PEREIRA, L.M., PEREIRA, F.C.N. and WARREN, D.H.D. (1979): User's Guide to DECSYSTEM-10 PROLOG, Occasional Paper No. 15, Department of Artificial Intelligence, University of Edinburgh.

RAMAMOHANARAO, K. and LLOYD, J.W. (1982): Dynamic Hashing Schemes, *Comput. J.*, 25, pp. 478-485.

RAMAMOHANARAO, K., LLOYD, J.W. and THOM, J.A. (1983): Partial-Match Retrieval using Hashing and Descriptors, *ACM Trans. on Database Syst.*, 8, (to appear).

REITER, R. (1981): Towards a Logical Reconstruction of Relational Database Theory, Department of Computer Science, University of British Columbia.

SAMMUT, C.A. and SAMMUT, R.A. (1983a): The Implemention of UNSW-PROLOG, *Austral. Comput. J.*, 15, pp. 58-64.

SAMMUT, R.A. and SAMMUT, C.A. (1983b): PROLOG: A Tutorial Introduction, *Austral. Comput. J.*, 15, pp. 42-51.

SANTANE-TOTH, E. and SZEREDI, P. (1982): PROLOG Applications in Hungary, in Clark and Tarnlund (1982), pp. 19-31.

TRELEAVEN, P.C. and GOUVEIA LIMA, I. (1982): Japan's Fifth Generation Computer Systems, *Computer*, August, pp. 79-88.

WARREN, D.H.D. (1981): Efficient Processing of Interactive Relational Database Queries Expressed in Logic, *Proc. of 7th International Conf. on Very Large Data Bases*, France, pp. 272-281.

WARREN, D.H.D. and PEREIRA, F.C.N. (1981): An Efficient Easily Adaptable System for Interpreting Natural Language Queries, Research Paper No. 155, Department of Artificial Intelligence, University of Edinburgh.

BIOGRAPHICAL NOTE

John Lloyd is a Lecturer in Computer Science at Melbourne University. He received his PhD in Mathematics from The Australian National University in 1973. Since then he has lectured in both Mathematics and Computer Science. Current research interests include logic programming and information retrieval.

The Implementation of UNSW-PROLOG

C. A. Sammut* and R. A. Sammut†

Early Prolog interpreters were based on techniques derived from the field of automatic theorem proving. These systems developed ways of implementing some of the more important features of the language, i.e. pattern directed invocation of procedures and automatic backtracking. Since the first Prolog system, a considerable amount of research has been devoted to creating more efficient interpreters and compilers. This paper discusses some of the more important aspects in the implementation of the Prolog interpreter, written and now used at the University of New South Wales.

Keywords and phrases: Prolog, logic programming, artificial intelligence, compiler design.
CR Categories: A.1, D.1, D.3, I.2.

1. INTRODUCTION

The field of Artificial Intelligence has always held an uncertain place in computer science. Often seen as separate from the mainstream of computing, AI has been an area in which many imaginative ideas have been born and studied. After years of refinement, these ideas gradually enter ordinary computing practice.

Automatic theorem proving has been under investigation for many years. However, its relevance to practical computing would seem doubtful. Nevertheless, the study of automatic theorem proving techniques has given rise to a number of new and promising programming languages. The most popular of these is Prolog.

Since writing Prolog programs is nothing like writing, say, Pascal programs, the user may well wonder how the interpreter could possibly work! This article is intended to help answer that question. If the reader is not familiar with Prolog, the companion article (Sammut and Sammut, 1983) gives an introduction to the language itself.

This exposition is based on an actual implementation: UNSW-Prolog, developed by the first author. It is written in the programming language 'C' and is intended to run under the Unix** operating system. UNSW-Prolog currently runs on VAX-11 and PDP-11 computers, and simpler versions have been ported to Z80, M6809 and Intel 8088 microprocessors. The system is largely compatible with DEC-10 Prolog (Warren, 1977) and much of the terminology used in this article is due to Warren.

The main feature which distinguishes UNSW Prolog from other DEC-10 Prolog interpreters from which it was developed is its convenient programming environment. Much of this advantage is derived from the Unix operating system. For example, when a program has been loaded into the Prolog system, the user may ask to edit a procedure without exiting Prolog. This is done by printing the definition of the procedure onto a temporary file, then invoking the text editor on the file. When the editing session is completed, the new definition is automatically reloaded and control returns to the Prolog interpreter.

Any Unix command may also be executed from within Prolog. This has been useful in developing a "help" system. Text explaining many features of UNSW-Prolog are stored in files in a "help" directory. When the user asks for help on a particular subject, Prolog looks up the appropriate file and displays it on the user's terminal.

Despite the simplicity of the UNSW-Prolog interpreter (or perhaps because of it) programs run quite quickly. The system has now been in use for teaching second year Computer Science and final year Electrical Engineering students for two years. It has proven to be reliable and relatively easy for the students to use.

In discussing the implementation of the Prolog interpreter, we will be particularly concerned with two features of the language: pattern directed invocation of procedures and automatic backtracking. Before considering these features, let us first look at the way in which a Prolog program is represented in the computer.

2. INTERNAL REPRESENTATION OF PROGRAMS

A Prolog program consists of a set of facts or clauses written in the form:

```
P :- Q, R, S.
```

P, Q, R and S are terms. P is called the 'head' of the clause and Q, R and S are the 'goals'. Terms may be atoms or compound terms of the form,

```
f(x, y, . . .)
```

where 'f' is the functional name (also called the principal term) and x, y, . . . are terms. Atoms may be words such as 'append' or 'fred'. They may also be numbers although numeric atoms must not appear as a head or goal in a clause.

In Prolog, programs and data structures are considered equivalent. That is, it must be possible to access the terms which make up a clause as if they were data. UNSW-Prolog achieves this by storing terms in a graph structure. As a simple example consider the representation for the term:

```
s(np(john), vp(v(likes), np(mary)))
```

The sentence 'John likes Mary' has been parsed into its component noun and verb phrases. The term can be represented by the tree shown in Figure 1. Each atom is stored

**Unix is a trademark of Bell Laboratories.

*Present address: Department of Mathematics and Computer Science, Saint Joseph's University, 5600 City Avenue, Philadelphia, PA, 19131, USA. †School of Electrical Engineering and Computer Science, University of New South Wales, Kensington, NSW, 2033. Manuscript received March 1983, revised April 1983.

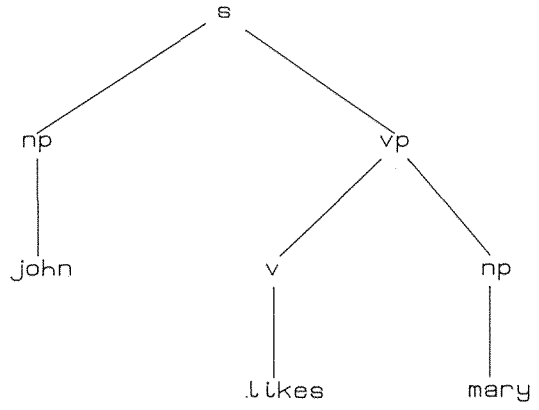


Figure 1. Terms are represented as trees.

in one node. The principal terms of a compound term form the roots of subtrees and the arguments go into the branches. Atomic terms are stored in the leaf nodes.

The UNSW-Prolog interpreter constructs this graph by allocating records from a heap. Each record corresponds to one node. The arcs between nodes are pointers in one record referring to another. Nodes may be words, numbers, compound terms, variables or clauses. Each record contains a type indicator to distinguish it from other kinds of records (Figure 2). The contents of the records are as follows:

ATOM – Symbolic atoms are words which contain three fields:
name: a pointer to a string of characters.
proc: if the atom is the name of a procedure, e.g. 'append', this field points to the first clause in the procedure.
link: a pointer used in hash table lookup.

INT – most Prolog systems implement only integer arithmetic.
int_val: the integer value of the number.

VAR – variables in each clause are numbered from 0 to N-1. N is the number of variables in the clause. These numbers are used as an offset into a stack which we will describe later. The print name of the variable is also stored so that clauses may be displayed while running Prolog interactively.

FN – Records for compound terms contain the *arity* of the term, i.e., the number of arguments. The arity is followed by a vector of pointers to terms. Term0 is the principal term, Term1 is the first argument, Term2 is the second, etc.

CLAUSE – *nvars:* the number of variables in the clause.

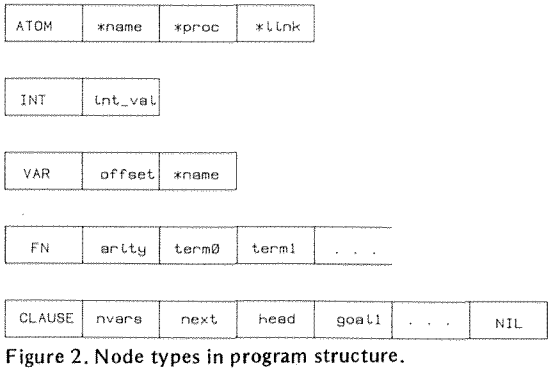


Figure 2. Node types in program structure.

next: a pointer to the next clause in the procedure.
head: a pointer to the head term.
goal1 . . .: a vector of pointers to the goals, terminated by a NIL pointer.

We can now see how a simple program for appending one list onto the end of another to produce a third is represented. The program is

```
append([], X, X).
append([A, ..B], X, [A, . .B1]) :– append(B, X, B1).
```

The internal structure of 'append' is shown in Figure 3. In the top left hand corner we see the record representing the atom 'append'. It contains a pointer to the first clause in the 'append' procedure. The first clause points in turn to the second clause. Each CLAUSE record has in it the number of variables used in the whole clause. The first one has only one variable, X, the second has four: A, B, X and B1.

Note the representation of the list [A, . .B]. Although lists are written differently from most other terms, this notation is only a convenience because lists are used so often. Internally, the list [1, 2, 3] is stored as if it had been written as

list(1, list(2, list(3, NIL)))

The list [A, . .B] is stored as if written:

list(A, B)

Now that we have all of these structures in memory what do we do with them? The single most common operation performed on Prolog data structures is pattern matching. In theorem proving, a particular kind of pattern matching algorithm, called 'unification' has been developed. This algorithm is described in the following section.

3. PATTERN MATCHING AND UNIFICATION

Unification is not just a simple pattern matching operation. It is the means by which Prolog builds the data structures which form the answers to the programmer's queries. This construction is achieved by means of the *logical variable.*

To illustrate the use of the logical variable, consider the following terms:

(1) s(np(john), vp(v(likes), np(Mary)))
(2) s(np(N), V)

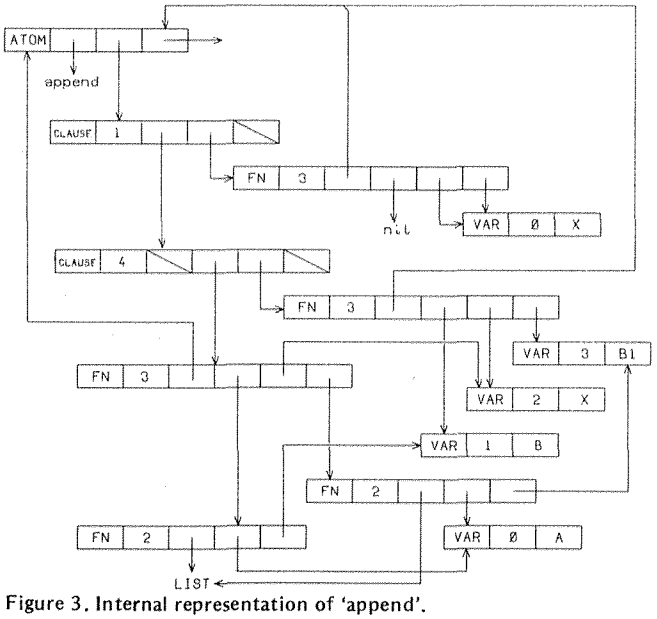


Figure 3. Internal representation of 'append'.

(3) s(N1, vp(V, N2))
(4) X

Suppose we want to unify (1) and (2).

(1) s(np(john), vp(v(likes), np(mary)))
(2) s(np(N), V)

To match a compound term, first match the principal terms. In this case, both terms have 's' as the principal terms. Then match the corresponding terms in the argument list. Thus we must match np(john) with np(N). Both terms are functions of 'np'. Next, match the arguments of 'np'. That is, match 'john' with 'N'. 'N' is, at present, an 'unbound variable'. When an unbound variable is matched with a constant term such as 'john' then a 'binding' is created between the variable and the other term. This will be denoted by {N/john}. Whenever, N is referred to from now on, we look up the set of bindings to find the value associated with N. In prolog, every term is associated with a binding environment or 'frame' such as the set above. In this example, there will be four frames corresponding to terms (1) to (4).

In UNSW-Prolog, frames are stored on a stack called the *variable stack*. Figure 4 shows the state of a variable stack after matching terms (1) and (2), terms (2) and (3) and terms (3) and (4), in that order.

The offset field in a variable record is used to give each variable a unique location within a stack frame. Thus when N becomes bound to 'john', a pointer to the atom representing john is placed in position 0 of the first stack frame (since N will be numbered 0). Similarly, when the two terms corresponding to the second arguments of terms (1) and (2) are matched, V will become bound to 'vp(v(likes), np(mary))'. Thus, a pointer to the term will be placed in frame offset 1.

Let us now try to match terms (2) and (3). Since term (3) contains variables, a new frame is added to the stack. This frame must be large enough to contain bindings for three variables.

(2) s(np(N), V)
(3) s(N1, vp(V, N2))

N1 becomes bound to np(N). However, to obtain the complete value of N1, it is necessary to know the value of N. The value to which N is bound is stored in the first frame. Therefore, the complete value of N1 is described by a pair consisting of a pointer to the term np(N) and another pointer to the frame in which N is bound. A term such as np(N) is called a 'skeleton' term because it forms the skeleton of a structure which will be built when the variable becomes instantiated (bound).

In order to find a match for the term vp(N, N2) we must get the value of V in term (2). This is done by looking at frame offset 1 in the frame for term (2). From this we see that V is bound to 'vp(v(likes), np(mary))'. Thus, V and N2 in term (3) become bound to v(likes) and np(mary) respectively.

A complete description of the unification algorithm is given below:

```
unify(term1, frame1, term2, frame2)
    if term1 = term2 then
        return (isatom(term) and isatom(term2))
            or (frame1 = frame2)
    if isinteger(term1) and isinteger(term2) then
        return term1.int_val = term2.int_val
    if isvariable(term1) then
        if isbound(term1, frame1) then
            return unify(termb, frameb, term2, frame2)
        else
            if isvariable(term2) and frame2>frame1 then
                return unify(term2, frame2, term1, frame1)
            else
                bind(term1, frame1, term2, frame2)
                return TRUE
    if isvariable(term2) then
        return unify(term2, frame2, term1, frame1)
    if iscompound(term1) and iscompound(term2) then
        if term1.arity <>term2.arity then
            return FALSE
        if term1.term[0] <>term2.term[0] then
            return FALSE
        for i: 1 . . term1.arity do
            if not unify(term1.term[i], frame1,
                         term2.term[i], frame2)
            then return FALSE
        return TRUE
    return FALSE
```

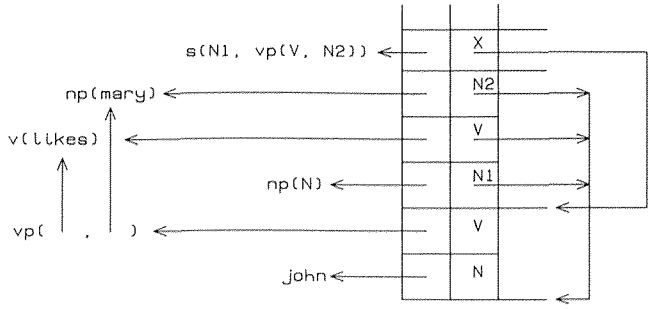


Figure 4. Variable stack during unification.

'Unify' returns true if the two terms are identical. When the terms are numeric, the function tests their integer values.

If term1 is a variable, we call the procedure 'isbound' which discovers if term1 has been bound to a value in frame1. If it has, the resulting value is assigned to 'termb' and the frame associated with that value is assigned to 'frameb'. Unification continues with these new values. If term1 is not bound then it may be bound to term2. However, if term2 is also a variable we may call unify again if frame2 > frame1. The reason for this will be explained later.

If term2 is a variable and term1 is not, unify is called with the arguments reversed. When both terms are compound terms, then unify first checks that they have the same arity. Then it makes sure that they have the same principal terms. Finally it enters a loop unifying all the corresponding arguments.

We have now seen how Prolog is able to construct data structures with the unification algorithm. The next step in building a Prolog interpreter is to understand the way in which the order of execution of goals is controlled. In particular we must know how automatic backtracking may be achieved.

4. CONTROLLING EXECUTION

As we mentioned earlier, Prolog has its origins in the field of automatic theorem proving. The satisfaction of a goal in Prolog is achieved by proving a "theorem" based on the logical clauses comprising the program. At the heart of any Prolog interpreter must, therefore be a procedure for determining the order in which goals are executed and the way in which the system selects clauses to be matched in order to construct the proof. The proof procedure in the UNSW-Prolog interpreter is based on the

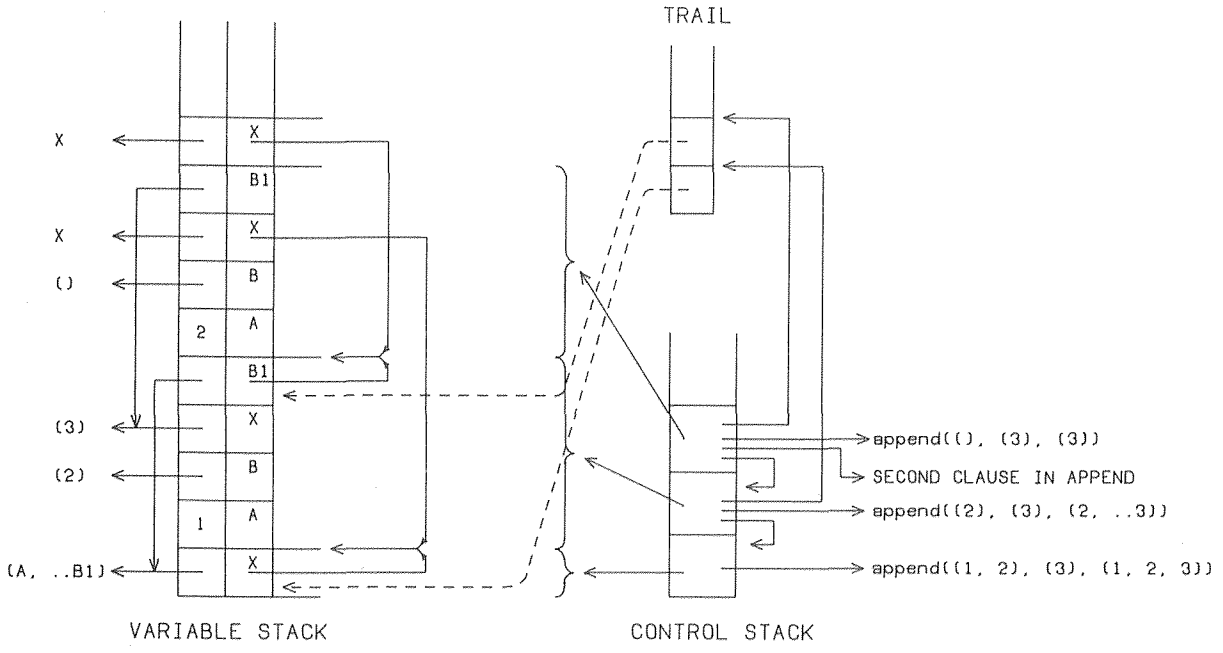


Figure 5. Snapshot of Prolog during execution of 'append'.

"lush" system of Kowalski (1974). The logical background to this system (and the historical origin of its name) are discussed by van Emden (1977). But now let us look at what is involved in implementing such a procedure.

The state of execution of Prolog is determined by a number of parameters.

1. Which goal in a clause is being executed?
2. Which frame on the variable stack is associated with the clause?
3. Which clause called the present clause? This must be known so that when a clause is completed successfully, execution can return to its caller.
4. What is the next clause in the currently executing procedure? If the present clause fails, Prolog must be able to abandon the execution of the clause and try the alternative clauses which define the procedure.
5. When all clauses in a procedure have failed, what should be tried next? Prolog must know where to backtrack to in order to attempt a new solution to a problem.
6. If backtracking becomes necessary, the bindings created by unification during the unsuccessful attempt must be undone. How do we find these bindings?

All of the information required to answer these questions is kept on another stack, the *control stack*. To execute a program, the Prolog interpreter sweeps through a structure such as the one described in Section 2. The unification algorithm is called to perform pattern matching and build data structures. As the interpreter performs its function, it leaves control information on the control stack. The contents of the stack are examined whenever the execution of a clause stops (because all goals have been satisfied or there has been a failure).

The interpreter requires yet another stack, the *trail*. Unification builds data structures by creating bindings on the variable stack. Each record on the variable stack is a pair of pointers, one to a skeleton structure stored in the heap, the other to a frame on the variable stack. These frame pointers may point to frames higher up or lower down on the stack.

When the execution of a goal fails, Prolog undoes the bindings created by cutting back the variable stack. This may leave some frame pointers lower down in the stack pointing up to invalid data. These pointers must be set to nil. Whenever a variable is bound to a term represented higher up in the stack, the address of the frame pointer which will be pointing upward is placed on the trail. When backtracking occurs, these pointers can be found on the trail and set to nil.

Remember that in 'unify' there was a test:

```
if isvariable(term2) and frame2 > frame1 then
    return unify(term2, frame2, term1, frame1)
```

This is to try to minimize the number of frame pointers which point upward.

Figure 5 shows the relationship between the three stacks during the execution of 'append'. We will discuss this diagram in detail after describing the procedure 'lush', the heart of the UNSW-Prolog interpreter.

A number of local variables will be used in 'lush':

sp: the top of stack pointer for the variable stack.
old_sp: saves sp for 'shallow backtracking'.
tp: the top of stack pointer for the trail.
old_tp: saves 'tp' for shallow backtracking.
env: the top of stack pointer for the control stack. This represents the most recent backtracking environment.
parent: a pointer to control stack record which contains the parent environment, i.e. the environment of the calling clause.
cl: a pointer to the vector of goals in a clause. This is a pointer to a pointer since the vector contains pointers to the goals.
t: the term pointer to which cl points.
frame1 and *frame2:* frame pointers used in unification.
clist: a pointer to the first clause which can match t.

```
lush(goal_list, number-of-variables)
    sp:= base of variable stack
    tp:= base of trail
    env:= base of control stack
    cl:= goal_list vector
    create a new frame on the variable stack large
        enough for 'number-of-variables'
    frame2:= sp
    old_tp:= tp
NEW_CLAUSE:
    frame1:= frame2
    parent:= env
NEW_GOAL:
    if first goal in cl vector is nil then goto SUCCEED
    t:= first goal in cl vector
    clist:= first clause which can match t
BACKTRACK_POINT:
    frame2:= sp old_sp:= sp
ALTERNATIVE:
    if clist is nil then goto FAIL
    create new frame large enough for clist.nvar variables
    if unify (clist.head, frame2, t, frame1) then
        save environment on control stack
        cl:= vector of goals in clist
        goto NEW_CLAUSE
    else
        sp:= old_sp
        set frame pointers on trail (as far as old_tp) to nil
        clist:= clist.next
        goto ALTERNATIVE
SUCCEED:
    if parent <> base of control stack then
        restore parent environment
        cl:= vector starting with next goal in goal list
        goto NEW_GOAL
    else print variables
FAIL:
    if env <> base of control stack then
        restore environment by popping control stack
        clear trail
        goto BACKTRACK_POINT
```

'Lush' can be viewed as a finite state machine. Each of the labels in the program represents a state. (This is a nice way of excusing the use of gotos!) The first task which must be completed is to prepare the interpreter to process a new clause. Once this is done, the next goal in the clause must be selected. At this point, the interpreter must also find the list of clauses which may match the goal. For example, if the goal is

```
append([1, 2], [3], X)
```

lush finds the clauses which describe 'append'.

Once these clauses have been found, lush will attempt to unify the goal with the head of each clause until a unification succeeds. There are several things to note at this point. Since the unification algorithm will try to match the current goal with a head of a clause, the system must have pointers to two frames: the frame for the goal, frame1, and the frame for the head, frame2.

If unification fails, lush will try to match the next clause in the list. When this happens, the environment must be reset to its state before the failed unification began. When that is done, the alternative clause is attempted. This is called 'shallow backtracking'. The control stack is not affected since all the information necessary for shallow backtracking is stored in the variables old_sp and old_tp.

If unification on all the clauses in a given procedure fails then the system must perform 'deep backtracking'. This is done by popping the last record on the control stack and using it to restore the environment of earlier state in the computation. An environment is saved when a goal successfully matches the head of a clause. The information stored in a control stack record consists of those parameters we listed at the beginning of this section.

Suppose we execute clause,

```
P :- Q, R, S.
```

Also suppose that Q has been executed successfully and Prolog now attempts to execute R. Note that when the system found a match for Q it recorded the environment *before* the match on the control stack. If R fails completely, then Prolog can return to Q and retry it, hoping that the alternative solution for Q will enable R to be satisfied.

Restoration of the environment takes place in the 'FAIL' state. After the restoration, lush goes to the BACKTRACK_POINT to restart the computation. Once unification has been successful, and the environment saved, Prolog begins processing the clause which matched the goal. This is done very simply. Since the calling or *parent* clause has been saved on the control stack, the system can reassign the variable 'cl' to point to a vector starting with the first goal in the new clause. The frame pointers and parent must also be updated.

An entire clause has been executed successfully when all the goals in the clause have been satisfied. This condition is recognized when 'cl', which is used to scan along the goals in a clause, points to the nil terminator placed at the end of all clauses.

When a clause has succeeded, control should return to the parent clause. The variable 'parent' points to the parent's environment record on the control stack. Thus lush can reconstruct the calling environment and continue executing the parent clause.

Note the difference between backtracking and returning to a parent. In the previous example, when R failed, records on the control stack were removed until Prolog could find an environment in which alternative solutions existed. If R succeeds, the parent's environment is restored without removing any records put on the stack during the execution of R. So if S fails, the computation of R can be restarted.

Once a parent environment has been restored, the parent clause is known to lush again. The pointer 'cl' is advanced to the next goal pointer and the procedure goes to the 'NEW_GOAL' state.

If a clause has succeeded, and it has no parent (i.e. 'parent' points to the base of the control stack) then Prolog has returned to the original query by the programmer. Thus, a complete solution has been found. At this point, the system can print the values of the unbound variables given in the query.

There may still be other possible solutions. These can be found by trying to match all possible alternative clauses left on the control stack. Remember, the environment records which store the alternative clause lists are only removed on backtracking. After printing one solution, lush falls through to 'FAIL' and begins backtracking to find the next solution. The system will continue to do this until all combinations of clause matches have been tried. When 'env' reaches the base of the control stack there are no more alternatives left and lush falls out of the bottom of the procedure and ends. We can view the execution of a Prolog program as depth first search through the space of all possible solutions to a problem.

A few details should be considered before finishing this section. Firstly, how are predefined predicates executed? It is possible to write many useful Prolog programs just using the interpreter described so far. However, it is usually necessary to add predefined predicates to perform operations such as arithmetic or input/output. In UNSW-

Prolog, predefined predicates are coded in 'C'. When a program has a goal such as 'print (X)' to output the term X, Prolog must find the code for 'print'. For 'C' code predicates such as 'print', the 'proc' field in the ATOM record points to a 'C' function rather than to a clause. There is a flag in the record to indicate the type of the pointer. When 'print' is encountered in the 'NEW_GOAL' state of lush, Prolog looks at the flag. Recognizing a predefined predicate, the system executes the 'C' function which performs the appropriate operation. If the function returns true then the next goal is selected and lush goes to the 'NEW_GOAL' state. If the function returns false, lush goes to 'FAIL' and backtracks.

One very important predefined predicate is the *cut* (denoted by '!'). The purpose of cut is to limit backtracking. In the clause,

P :- Q, R, !, S.

If S fails then the execution of the entire clause will be aborted and Prolog will not try to find an alternate clause to match. The cut following R instructs the interpeter to 'forget' any alternative solutions to the procedure it is executing.

Cut can be implemented quite simply. A cut indicates that all the backtrack points encountered while executing the current procedure should be ignored. Therefore, the control stack pointer, 'env', can be reset to point to the parent environment. So when a failure occurs, backtracking will occur in the calling clause, ignoring other possibilities in the procedure called.

The interpreter described here was designed with simplicity as its main goal. Lush, unify and the other procedures which manipulate the various stacks can be written in only four pages of 'C' code. Despite its simplicity, UNSW-Prolog is fast. However, there are a number of features which may be added to the system to improve its performance, particularly in reducing the amount of information held on the variable stack. These additions will be discussed in Section 6 after we have looked at a complete trace of execution of

? append([1, 2], [3], X).

5. A TRACE OF EXECUTION

Figure 5 illustrates the relationship between the three stacks in UNSW-Prolog. The diagram shows a snapshot of the system when 'append' has completed its task in building the list X. The procedure has been called (recursively) three times. The control stack has three records which were created at each call. Each record contains:

- pointers to the frame created by the unification,
- a pointer to the trail so that bindings may be undone if backtracking is necessary,
- a pointer to the control record of the parent environment,
- a pointer to the goal being executed,
- a pointer to alternative clauses to match. (This is nil in the bottom two records because the second clause is active. The top record contains a pointer to the second clause since the first clause is active.)

To show how this structure was built up, we reproduce in Figure 6 a trace of execution produced by UNSW-Prolog. The trace consists of a dump of the contents of the variable stack and control stack. The trail has been omitted.

The variable stack dump shows <frame pointer/skeleton> pairs numbered from 0 upward. Frame pointers are displayed as offsets from the bottom of the stack. The value of a binding may be read by looking at the skeleton term and filling in the values of the variables. These values are found by looking up the binding at the offset into the frame indicated.

The control stack shows,

- the number of the parent record (−1 is the bottom of the stack),
- the base of the frame for that environment and the base of the next frame above,
- the trail pointer,
- the goal associated with the environment.

The alternative clauses are not shown.

```
------------- VARIABLE STACK -------------
10 :
 9 : 5    X̄
 8 : 0    [3]
 7 : 1    X
 6 : 0    []
 5 : 0    2
 4 : 5    [A, ..B1]
 3 : 0    [3]
 2 : 0    [2]
 1 : 0    1
 0 : 1    [A, ..B1]

------------- ENVIRONMENT STACK ------------
 2 :    1    5    9    2    append([], [3], [3])
 1 :    0    1    5    1    append([2], [3], [2, 3])
 0 :   -1    0    1    0    append([1, 2], [3], [1, 2, 3])
==========================================

------------- VARIABLE STACK -------------
10 :
 9 : 5    X̄
 8 : 0    [3]
 7 : 1    X
 6 : 0    []
 5 : 0    2
 4 : 5    [A, ..B1]
 3 : 0    [3]
 2 : 0    [2]
 1 : 0    1
 0 : 1    [A, ..B1]

------------- ENVIRONMENT STACK ------------
 2 :    1    5    9    2    append([], [3], [3])
 1 :    0    1    5    1    append([2], [3], [2, 3])
 0 :   -1    0    1    0    append([1, 2], [3], [1, 2, 3])
==========================================

------------- VARIABLE STACK -------------
10 :
 9 : 5    X̄
 8 : 0    [3]
 7 : 1    X
 6 : 0    []
 5 : 0    2
 4 : 5    [A, ..B1]
 3 : 0    [3]
 2 : 0    [2]
 1 : 0    1
 0 : 1    [A, ..B1]

------------- ENVIRONMENT STACK ------------
 2 :    1    5    9    2    append([], [3], [3])
 1 :    0    1    5    1    append([2], [3], [2, 3])
 0 :   -1    0    1    0    append([1, 2], [3], [1, 2, 3])
==========================================

------------- VARIABLE STACK -------------
10 :
 9 : 5    X̄
 8 : 0    [3]
 7 : 1    X
 6 : 0    []
 5 : 0    2
 4 : 5    [A, ..B1]
 3 : 0    [3]
 2 : 0    [2]
 1 : 0    1
 0 : 1    [A, ..B1]

------------- ENVIRONMENT STACK ------------
 2 :    1    5    9    2    append([], [3], [3])
 1 :    0    1    5    1    append([2], [3], [2, 3])
 0 :   -1    0    1    0    append([1, 2], [3], [1, 2, 3])
==========================================
```

Figure 6. Trace of execution.

The first call to 'append' occurs when the query

```
? append([1,2],[3],X)
```

is put to the system. This results in a match on the second clause. Note that unbound variables are displayed as underscores '_'. The frame constructed contains only the one variable, X. The second clause is entered and the only goal

```
append(B,X,B1)
```

is matched again with the head of the second clause. Note that X in the first environment has been partially bound to [1,..._]. This process is repeated until the first clause,

```
append([],X,X).
```

is matched. At this point, all of the variables have been bound to their final values.

6. IMPROVING EFFICIENCY

The most obvious way of gaining a significant increase in the speed of Prolog is to build a compiler which could generate machine code to execute programs. A discussion of such a compiler would require a much longer article. Instead, the reader is referred to Warren (1977).

There are a number of ways in which an interpreter can be made to use space more efficiently. The reader is invited to construct a trace of execution for the program:

```
reverse([],[]).
reverse([A,..B],X) :- reverse(B,B1),append(B1,[A],X).
```

This program finds the reverse of a list whose head is A and whose tail is B by reversing B to produce B1 and appending [A] onto the end of B1.

Remember that 'append' creates an entirely new list. So in the course of executing 'reverse' Prolog will build and throw away temporary list structures. There is a problem: since the variable stack is cut back only when backtracking occurs and all of these temporary lists are built on the variable stack, a lot of unwanted information will remain to occupy space on the stack. A more efficient version of reverse which does not use 'append' can be written, however, this example is used to illustrate a general problem.

It is possible to add a 'garbage collector' to Prolog. When the variable stack grows too large, the garbage collector sweeps through the stack looking for frames which consist completely of variables which are unused. This is quite a complex operation since the collector must first find and mark all the bindings that are in use. Once this is done, the active frames must be compacted towards the bottom of the stack, making sure that all the pointers to the frames have been updated to point to the new locations.

One interesting feature of Prolog is that variables in the head of a clause may be used for either input or output. Consider the program,

```
in(X,[X,..Y]).
in(X,[A,..B]) :- in(X,B).
```

If a query such as

```
? in(X,[1,2,3])
```

is put to the system, X will be bound to each member in turn of the list [1, 2, 3]. If the query

```
? in(1,X),!.
```

is issued then X will be bound to a list of which 1 is a member. (What would happen if the cut were omitted?)

In the case of procedures such as 'append', we usually use them in only one way, i.e. the first two arguments are inputs and the third an output. This means that in the second clause, variables B and X are not used in the construction of an output. However, they will remain on the stack, even with a garbage collector, because the other variables, A and B1, are used to construct an output.

There are two ways around this problem: If Prolog knows in advance which variables will be used to construct an output and which will be used only as local variables then two variable stacks may be maintained. Output variables (or global variables) will be placed in the ordinary variable stack. Remember that this stack is only cut back on backtracking. Local variables will be put on a separate local stack. This can be cut back whenever a procedure completes deterministically, i.e. when there are no more solutions left. In practice, the local stack can be combined with the control stack since they both grow and shrink together. This technique is used in DEC-10 Prolog (Warren, 1977).

The other method, due to Bruynooghe (1982) requires a completely different way of building Prolog data structures. The reader is referred to the paper for a detailed description. This approach treats all variables as local variables. Whenever a variable is bound to a compound term, a complete copy of the term is put onto a *copy* stack. This keeps the variable stack small at the expense of the time required to do the copying.

7. CONCLUSION

An important attribute of Prolog is that the implementation of such a powerful language need not be inefficient. UNSW-Prolog runs in about 50k bytes of memory on the PDP-11/70. Of this, half is shareable code.

Another attribute of the language is that it is easy to use. Because of the simplicity of the language, the beginning programmer needs to adjust to relatively few new concepts. However, ease of use is also an attribute of the specific implementation. In this aspect, the Unix operating system helps the implementor considerably.

For example, UNSW-Prolog allows programmers to change the definition of a predicate without having to exit from the Prolog session. This is done by 'pretty printing' the definition of the predicate onto a temporary file and then forking an editor program to work on the file. When the changes to the predicate have been made, the new definition is read back in automatically and replaces the old program. All of this can be coded in just a few lines of 'C'.

In this article we have seen how the main features of Prolog can be implemented. The system provides capabilities not found in most programming languages consequently, Prolog interpreters must use techniques not normally found in the 'tool kit' of language designers. Although Prolog is different, it need not be inefficient. As logic programming gains in popularity, implementations will continue to improve in performance.

8. REFERENCES

BRUYNOOGHE, M. (1982): The memory management of Prolog implementations, in *Logic Programming* (eds. K.L. Clark and S.-A. Tarnlund), Academic Press, London, pp. 83-98.

van EMDEN, M.H. (1977): Programming with resolution logic, *Machine Intelligence 8* (eds. Elcock, E.W. and Michie, D.), Ellis Horwood, Chichester, pp. 266-299.

KOWALSKI, R.A. (1974): Predicate logic as programming language, *Proc. IFIP74*, North Holland, pp. 569-574.

SAMMUT, R.A. and SAMMUT, C.A. (1983): PROLOG: A tutorial introduction, *Austral. Comput. J.*, 15, pp. 42-57.

WARREN, D.H.D. (1977): *Implementing PROLOG: compiling predicate logic programs*, Department of Artificial Intelligence Research Report No. 39, University of Edinburgh.

BIOGRAPHICAL NOTE

See the accompanying paper "PROLOG: A Tutorial Introduction" by the same authors in this issue.

Hidden Arcs of Interpenetrating and Obscuring Ellipsoids

By D. Herbison-Evans*

The drawing of three-dimensional outlines of objects composed of ellipsoids is considered. Any outline may be obscured because either it is behind another ellipsoid, or it penetrates into another ellipsoid. The outline to be drawn is divided into arcs by the points where it intersects the outline or penetrates the body of an obscuring ellipsoid. Each arc is such that if part of it is hidden then it is all hidden. In order to discover which arcs are hidden, they must be tested.

This paper discusses the use of topology to reduce the number of arcs to be tested. The number drops from eight to three in the most complex case, and is halved in most of the other cases. The use of the topology is complicated by the unpredictable zero point of the outline. The testing scheme presented here accommodates this problem. It is encapsulated in a number of tables in the paper.

Keywords and phrases: Hidden lines, solid modelling, quadric surfaces, quartic equations.
CR Categories: D.2.2, I.3.5, I.3.7.

Introduction

This problem arises in trying to draw, by computer, outlines of figures composed of three dimensional ellipsoids. The outlines of the ellipsoids are two dimensional ellipses. It is desirable to omit those parts of the ellipse outlines which are hidden by other ellipsoids (Herbison-Evans, 1978). The type of figure being considered is shown in Figure 1.

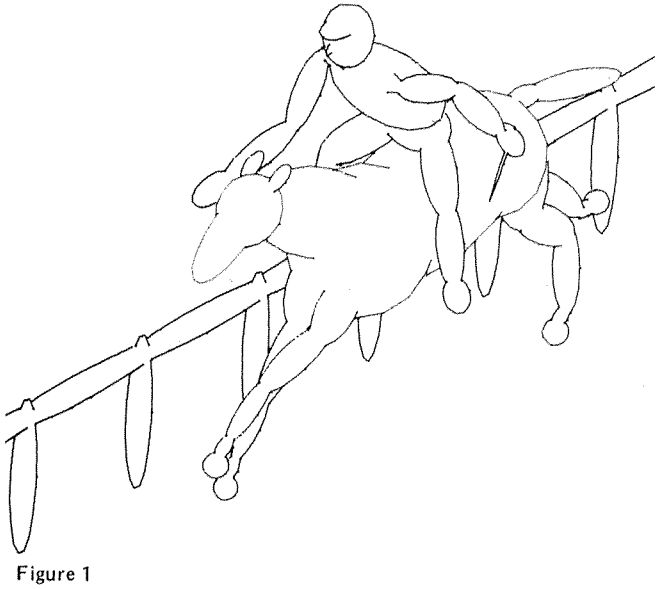


Figure 1

Visible Arc Endings

The visible arcs of an ellipse outline may disappear at either of two sorts of points:

(a) Obscuration: where the outline goes behind the outline of another ellipsoid,

(b) Penetration: where the outline goes into another ellipsoid.

The points of obscuration are where the outline of the drawn ellipsoid intersects the outline of the obscuring ellipsoid.

The points of penetration are more complex. The outline of the drawn ellipsoid lies in a plane. This plane slices through the obscuring ellipsoid, possibly revealing an elliptical section. This may be projected onto the viewing plane to give another ellipse. It is the intersections of this ellipse with the drawn ellipse which may be penetration points for the drawn outline. These ellipsoids, ellipses and points are illustrated in Figures 2, 3 and 4.

In both cases, the points are those resulting from the simultaneous solution of two ellipse equations: two outlines for the obscuration points, and an outline and a section for the penetration points. In each case, the simultaneous quadratics may be reduced to a single quartic equation which may be solved analytically (Herbison-Evans, 1982).

Either 0, 2 or 4 roots are obtained from each quartic equation. Thus the drawn outline may be divided into up to eight arcs by these points. Each arc has the property that if any point on that arc is hidden, then the whole arc is hidden. In order to discover whether an arc is visible, the (x, y) co-ordinates of its midpoint may be inserted into the equation for the obscuring ellipsoid. The z depths of the drawn outline and the nearer surface of the obscuring ellipsoid then govern the visibility of the arc. These tests are time consuming, taking between 15 and 18 multiplications (and similar numbers of additions and house-keeping operations) per test. As it is desirable to obtain real-time animation speeds on cheap equipment (Herbison-Evans, Green and Butt, 1982), the number of such operations must be minimised.

Topology

The two sorts of points are constrained by the figures from which they are derived. This becomes clearer when the points are ordered. If the ellipse is viewed as a distorted circle, then an angle running from 0 to 2π can specify the outline of the ellipse (Cohen, 1971). Let each intersection point be specified by a parameter in the range 0 to 2π.

*Basser Department of Computer Science, University of Sydney, Sydney, NSW 2006. *Manuscript received August 1982, revised February 1983.*

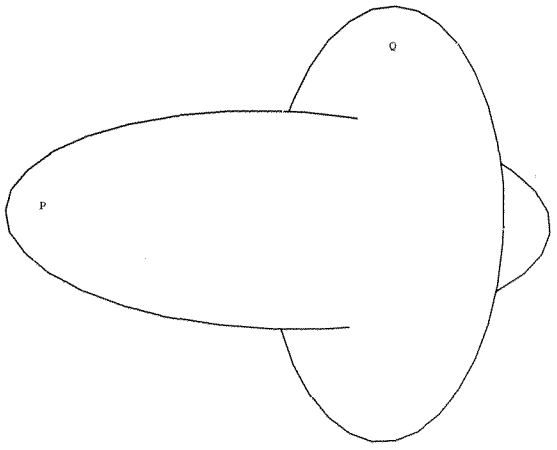


Figure 2. Two interpenetrating ellipsoids P and Q viewed from front.

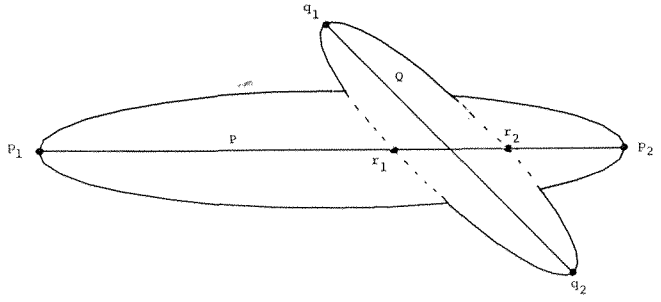


Figure 3. Two interpenetrating ellipsoids P and Q viewed from above. p_1p_2 = plane of outline of ellipsoid P viewed from front. q_1q_2 = plane of outline of ellipsoid Q viewed from front. r_1r_2 = slice of (p_1p_2) plane through ellipsoid Q.

Then the obscuration points (a1, a2, a3 and a4) and the penetration points (b1, b2, b3, b4) may be sorted into ascending order (assuming there are four of each). They will then be found to occur cyclically in pairs. For instance let a1 be the lower of a pair of obscuration points which have no penetration points between them. Without loss of generality, subtract a1 from all eight points, and add 2π to any that go negative in this process. Then the final order will be:

a1, a2, b1, b2, a3, a4, b3, b4.

Then the arcs b1-b2 and b3-b4 will always be hidden. The arcs a1-a2 and a3-a4 will always be visible. Only the arcs a2-b1, b2-a3, a4-b4 and b4-a1 need testing. Of these, an even number must be visible, thus only three require testing. The fourth may be assigned by parity on the other three.

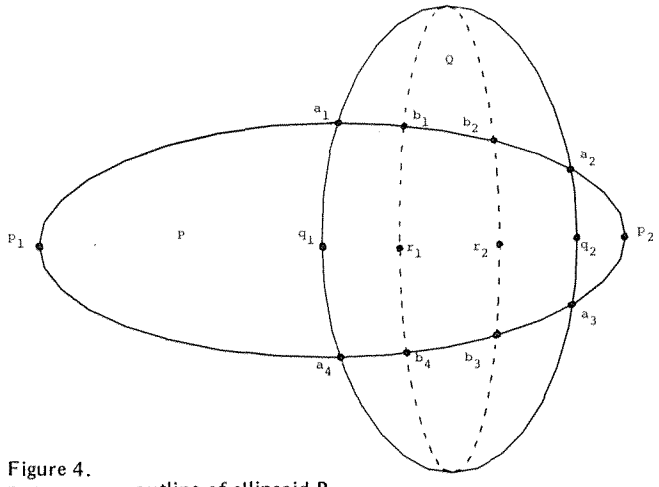


Figure 4.

p_1p_2	outline of ellipsoid P.
q_1q_2	outline of ellipsoid Q.
r_1r_2	projection onto viewing plane of ellipse of intersection of outline plane of P with ellipsoid Q.
$a_1a_2a_3a_4$	intersections of outline of ellipsoid $P(p_1p_2)$ with outline of ellipsoid $Q(q_1q_2)$.
$b_1b_2b_3b_4$	intersections of outline of ellipsoid $P(p_1p_2)$ with projection (r_1r_2) of intersection of plane of ellipsoid P with ellipsoid Q.

Similar constraints apply when there are two obscurations or penetration points instead of four.

Practical Considerations

It is desirable to work in terms of the tangent of half the angular parameter described above. This minimises the number of operations in performing an obscuration test. It means that the parameter runs from −infinity to +infinity instead of 0 to 2π.

The arbitrary zero point of the parameter may lie within any of the arcs. Symmetry reduces this to four classes when there are four roots of each type. The classes may be identified by referring to the list of points sorted into ascending order. The index of the lowest obscuration point (a1 or a2) that is immediately above the second lowest penetration point (b2) separates classes 1 and 2. Classes 3 and 4 may be separated by whether the lowest penetration point (b1) is below or above the second lowest obscuration point (a2).

Each class may have those arcs tabulated that are always hidden, those that need testing for being hidden, and those that are also hidden if the tested arcs are indeed hidden. This is shown in Table 1. Note that the fourth arc (in brackets) whilst it is specified to be tested, need only

Table 1. Hidden and Testable Arcs of an Ellipsoid Outline with Four Intersections and Four Penetrations.

Class	Point Sequence	Test	c1	c2	c3	c4	c5	c6	c7	c8
1	b1 b2 a1 a2 b3 b4 a3 a4	b2 < a1	a4	+∞	−∞ a2	b1 b3	b2 b4	a1 (a3)		
2	a1 b1 b2 a2 a3 b3 b4 a4	b2 < a2			a1 a3	b1 b3	b2 b4	a2 (a4)		
3	a1 a2 b1 b2 a3 a4 b3 b4	a2 < b1			a2 a4	b1 b3	b2 b4	a3 (+∞)	−∞	a1
4	b1 a1 a2 b2 b3 a3 a4 b4	other			a2 (a4)	b2 −∞ b4	b3 b1 +∞	a3 a1		

Table 2. Hidden and Testable Arcs of an Ellipsoid Outline with Four Intersections and Two Penetrations.

Class	Point Sequence	Test	c1	c2	c3	c4	c5	c6	c7	c8
1	b1 b2 a1 a2 a3 a4	b2 < a1	a4	+∞	−∞	b1 a2	b2 a2	a1 a3		
2	a1 b1 b2 a2 a3 a4	b2 < a2			a1	b1 a3	b2 a3	a2 a4		
3	a1 a2 b1 b2 a3 a4	b2 < a3			a2	b1 −∞	b2 −∞	a3 a1	a4	+∞
4	a1 a2 a3 b1 b2 a4	b2 < a4			a3	b1 a1	b2 a1	a4 a2		
5	a1 a2 a3 a4 b1 b2	b1 > a1			a4	b1 a2	b2 a2	+∞ a3	−∞	a1
6	b1 a1 a2 a3 a4 b2	Other			a4	−∞ b2 a2	b1 +∞ a2	a1 a3		

Table 3. Hidden and Testable Arcs of an Ellipsoid Outline with Two Intersections and Four Penetrations.

Class	Point Sequence	Test	c1	c2	c3	c4	c5	c6	c7	c8
1	a1 a2 b1 b2 b3 b4	a2 < b1			a2	b1 b3	b2 b4	b3 +∞	−∞	a1
2	b1 a1 a2 b2 b3 b4	a2 < b2			a2	b2 b4 −∞	b3 +∞ b1	b4 a1		
3	b1 b2 a1 a2 b3 b4	a2 < b3	a4	+∞	−∞ a2	b1 b3	b2 b4	a1		
4	b1 b2 b3 a1 a2 b4	a2 < b4			a2	b4 −∞ b2	+∞ b1 b3	b2 a1		
5	b1 b2 b3 b4 a1 a2	b4 < a1	a2	+∞	−∞	b1 b3	b2 b4	b3 a1		
6	a1 b1 b2 b3 b4 a2	Other			a1	b1 b3	b2 b4	b3 a2		

Table 4. Hidden and Testable Arcs of an Ellipsoid Outline with Two Intersections and Penetrations.

Class	Point Sequence	Test	c1	c2	c3	c4	c5	c6	c7	c8
1	b1 b2 a1 a2	b2 < a1	a2	+∞	−∞	b1	b2	a1		
2	a1 b1 b2 a2	b2 < a2			a1	b1	b2	a2		
3	a1 a2 b1 b2	a2 < b1			a2	b1	b2	+∞	−∞	a1
4	b1 a1 a2 b2	Other			a2	b2 −∞	+∞ b1	a1		

have the parity of the number of previous hidden arcs tested. Of the four potential hidden arcs, only a total of 0, 2 or 4 may actually be hidden. Thus it is hidden if the previous three tests showed 1 or 3 hidden arcs.

In the table, the points marked c1 to c8 have the following meanings:

c1-c2	hidden if c3-c4 hidden
c3-c4	requires testing
c4-c5	always hidden
c5-c6	requires testing
c7-c8	hidden if c5-c6 hidden

4-2 Case

When there are four obscuration points and two penetration points, there is less symmetry. There are six different positions of the origin. These six positions may be classified by the same algorithm as in the 4-4 case: by consideration of the sorted list of points. Classes 1, 2, 3 and 4 may be identified by the index of lowest obscuration point immediately above the second lowest penetration point (b2). Classes 5 and 6 may be separated by whether the lowest penetration point (b1) is below or above the lowest obscuration point (a1).

In all classes, three arcs must be tested. They are listed in Table 2, with the same key as in Table 1.

2-4 Case

As in the 4-2 case, there are six classes for the location of origin. Classes 1, 2, 3 and 4 may be identified when the points are in ascending order, by the index of the penetration point that is immediately above the higher obscura-

tion point (a2). Classes 5 and 6 may be distinguished by whether the lower obscuration point is below the first penetration point.

For all classes, two of the b-b arcs are always hidden. The arc between them, and between their ends and each of the a points need testing. The a-a arc is always visible. Thus, again three arcs need testing.

2-2 Case

In this case, the b-b arc is always hidden, the a-a arc always visible, and the two a-b arcs require testing. Each may separately be hidden. There are four classes depending on the position of the origin. They are separated as with the 4-4 and 4-2 cases. The results are tabulated in Table 4.

4-0 Case

In this case, 0 or 2 arcs must be hidden, and they must be non-adjacent. One arc may be chosen arbitrarily and tested for being hidden. This test may also test the point for the inclusion of its projection in the projected outline of the obscuring ellipsoid at little extra cost of computation. If either is true, then the two adjacent arcs need testing. Thus two or three tests may need to be performed.

2-0 Case

Either 0 or 1 of the two arcs can be obscured. If the first arc selected for testing is hidden or is within the obscuring outline, then no other test is required. If it is neither, then the other arc needs testing. Thus either one or two tests are performed.

0-4 Case

Of the four arcs, two must be obscured. They must be opposite (non-adjacent) arcs. Only one arc need be tested to discover which pair are obscured.

0-2 Case

Either or both of the arcs may be hidden, so two tests are required for this case.

Conclusion

The techniques described here cut down the number of arcs to be tested for being hidden when drawing interpenetrating ellipsoids. The improvement is from eight to three in the most complex case. Most cases have the number of arcs to be tested halved, which halves the time for the calculation of the hidden arcs in polyellipsoid scenes.

One problem with these techniques is the reliance on the obscuration and penetration point values for deriving the topology. If any of the values is in error by an amount which moves it past one of the other values, the topology will be nonsense or wrong.

A nonsense topology can be detected by incorrect pairing of roots e.g., abab in the 2-2 case. In a nonsense case, one can revert to testing all of the arcs. This will slow the system down briefly but if the error is small, will not much affect the drawing. Hopefully, it will not happen often enough to slow the system noticeably.

If the values are wrong in such a way as to still give a valid (but incorrect) topology, the system will assign erroneous visibilities to the arcs. It will cause the drawing to appear obviously incorrect. If the frequency of this problem is disturbingly high, then the slower system (testing all the arcs) must be used. It is more robust in the presence of errors. As in most situations: speed has its price.

References

HERBISON-EVANS, D. (1978): "NUDES2: A Numeric Utility Displaying Ellipsoid Solids, Version 2, *Computer Graphics*, Vol. 12, No. 3, pp. 354-356.

HERBISON-EVANS, D. (1982): The Inaccurate Solution of Cubic and Quartic Equations, Basser Department of Computer Science, Sydney University, Technical Report 187.

HERBISON-EVANS, D., GREEN, R.D. and BUTT, A. (1982): Computer Animation with NUDES in Dance and Physical Education, *Austral. Comput. Sci. Commun.*, 4, No. 1, pp. 324-331.

COHEN, D. (1971): On Linear Difference Curves, R.D. Parslow and R.E. Green (eds.), *Advanced Computer Graphics*, Plenum Press, pp. 1143-1177.

BIOGRAPHICAL NOTE

Don Herbison-Evans is a senior lecturer in the Basser Department of Computer Science at the University of Sydney, NSW. His research interests cover the fields of numerical analysis, computer graphics and animation, dancing, and holograms. He is chairman of the Australasian Computer Graphics Association. He is also on the national council of the Association for Computer Aided Design, is a fellow of the Royal Astronomical Society and the Royal Society of Chemistry, and is a member of the ACM, IEE, IEEE, and the Australian Computer Society.

Herbison-Evans received an honours BA and a PhD in chemistry from Oxford University in 1961 and 1963, respectively.

Application of Structured Design Techniques to Transaction Processing

By I. T. Hawryszkiewycz* and D. W. Walker*

The limitations of structured design in a transaction processing environment are discussed. In particular, a conflict between top down hierarchical control, inherent in structured design, and the bottom level activation assumed in transaction processing is noted. Three alternative methods for resolving this conflict so that structured design reductions can be adapted to the transaction processing environment are then discussed.

Keywords and phrases: structured design, top-down design.
CR Categories: D.1, D.2.

INTRODUCTION

Structured analysis (Gane and Sarson, 1978, de Marco, 1978) and structured design techniques (Yourdon and Constantine, 1979) have been found effective in the design of computer-based information systems. Their two most well-known advantages are that

- structured analysis techniques are an effective way of precisely and unambiguously describing both existing and proposed systems in a way understood both by users and programmers, and
- structured design defines structured (rather than ad-hoc) methods for converting proposed systems to a computer implementation.

These two steps differ in one respect. Structured analysis is independent of its environment. Its techniques need not be in any way modified or tailored to a particular user problem. Structured design on the other hand must package the structured specification into the available computer facilities. The design techniques must often be tailored to this environment. This paper describes the tailoring necessary to accommodate structured design to transaction processing systems. The paper first describes the constraints imposed by transaction processing software on structured design. Such software is supplied by manufacturers and includes systems such as Honeywell's TDS or IBM's CICS. This software often controls module activations and parameter passing. Structured design must then be adapted to the operation of this software. Methods of doing so are discussed in this paper.

Structured Design

Structured design proceeds in the stages illustrated in Figure 1. The designer commences with the structured specification, which is made up of

- a set of data flow diagrams,
- a set of mini specifications usually written in structured or tight English, and
- a data dictionary.

The first step of structured design is to convert the data flow diagram into a structured chart using transform and transaction analysis (Yourdon and Constantine, 1979). In summary, the conversion identifies transform and transaction centres within data flow diagrams. As shown in Figure 2, the transform centre is the central process or transformation in the data flow diagram; it has a well defined input and output streams. A main module is created in the structure chart for the transform centre. The main module activates modules that correspond to the central process and its output and input processes. The data flow input processes become the input modules (often called the afferent branch). The output processes become the output modules (often called the efferent branch). The modules in the structure chart are chosen to satisfy a number of criteria generally classified into coupling and cohesion criteria. Coupling defines the kind of interconnection between modules. The best coupling is defined to be

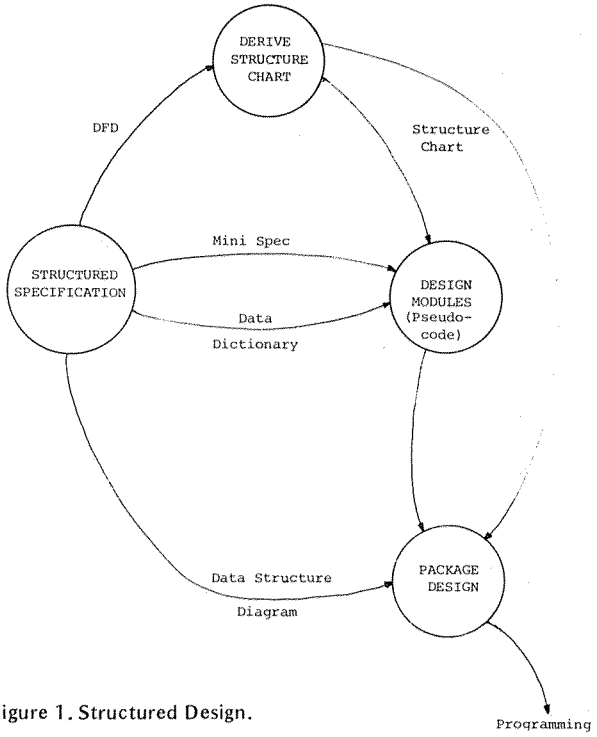


Figure 1. Structured Design.

*School of Information Sciences, Canberra College of Advanced Education. Manuscript received September 1982, revised March 1983.

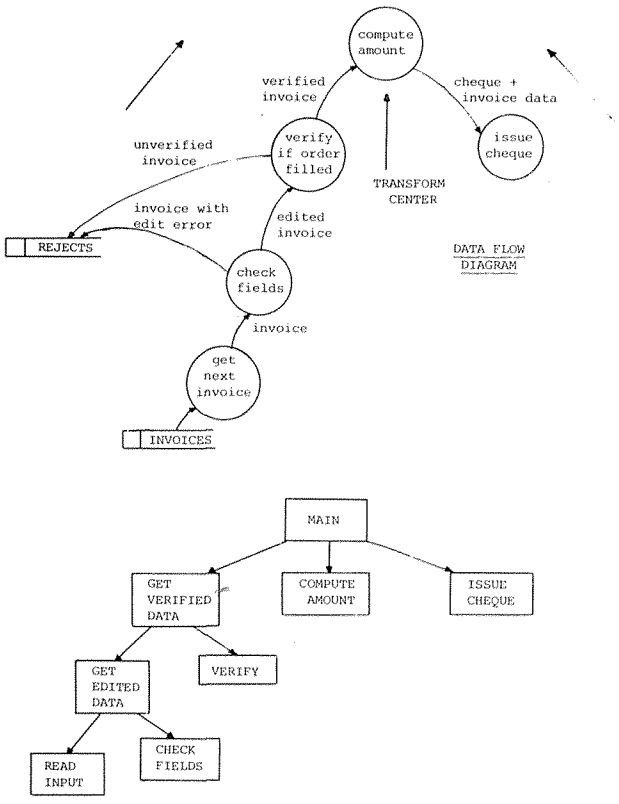


Figure 2. Transform Analysis.

data coupling where only data elements of common interest are passed as arguments between two modules. Less desirable forms of coupling are

- a control coupling where control data (for example, flags), which affects module execution, is passed between modules,
- a common environment coupling, where modules share data in a common area, or
- content coupling where one module can modify the internal data or statements in another module or branch into that module.

Cohesion on the other hand concerns the reason for associating code in the same module. The most desirable cohesion is known as functional cohesion, which implies that the module carries out one well-defined function. Less desirable forms of cohesion are defined by Yourdon and Constantine (1979) to be

- communicational, where the same module contains processes with the same input or output,
- procedural, which results from the subdivision of a flowchart,
- temporal, where each module includes functions which are related in time,
- logical, where all functions are of the same class (for example, all edit functions in one module), or
- coincidental, where no meaningful relations exist between the module functions.

The transform reduction in Figure 2 results in modules that are data coupled and functionally cohesive.

Once the broad module structure of the program is chosen, the next two steps in Figure 1 are to design the modules and package them into load modules. Module design is usually based on the mini-spec. Each mini-spec defines the function of a process in the data flow diagram. This process must then be realized by the corresponding process module. This is usually done in a number of steps. First, the mini-specs are converted to pseudo-code, which becomes the module specification. The modules are then packaged into load units supported by the available computer system. Once this is done detailed programming can commence.

The fundamental premise of transform or transaction analysis is that all input and output is controlled from the main module. Thus in Figure 2 the MAIN module calls GET-VERIFIED-DATA to get input A. This call is passed down the afferent branch until module READ-INPUT which reads a card. The card image is then passed to module CHECK-FIELDS for editing. If the edit checks are passed, then the edited message is returned to GET-EDITED-DATA and then passed to GET-VERIFIED-DATA and then to the VERIFY module. If verification succeeds then the message is passed to MAIN through module GET-VERIFIED-DATA. A failure at any stage will return control to READ-INPUT module.

This execution mode, however, is inconsistent with most transaction processing software. In Figure 2 input is initiated by the main program; in transaction processors, however, it is the arriving input message that initiates the program. The arrival of a message of a given type activates a predefined module for that type.

Ideally, given the structure of Figure 2, an incoming transaction message should activate the lowest level module, namely, READ-INPUT. However, a return cannot be made to the MAIN module in the same way as assumed in the standard structure as in this case the MAIN module has not been activated. Alternatively, a message can activate the MAIN module. However, activation of main modules by an input message violates many of the coupling and cohesion criteria. For example, the whole message would first be passed down the afferent branch to the lowest module. Returns would then be made up the branch with flags set to indicate the success achieved at each level of input. The modules would then become dysfunctional as each module in addition to performing its function would now also act as a means of passing messages. The coupling now takes a control nature as flags indicating the success of verification or editing must accompany the message or return to MAIN. The control information was not in the batch mode as only correctly edited and verified messages were returned to MAIN.

The problem then is to reconcile structured design with transaction processing. One would like to apply the structured design techniques in a transaction driven environment. Some methods of doing this are discussed in the remainder of the paper. Before doing this the transaction processing environment is discussed in more detail.

The Transaction Processing Environment

Transaction processors provide programmers with facilities to control message flow through a software system. The characteristics of transaction processors have been described in earlier literature (e.g. Clarke, 1982). Transaction processors usually include facilities to

- activate modules depending on the value of some field in an incoming message,
- direct messages between program modules,

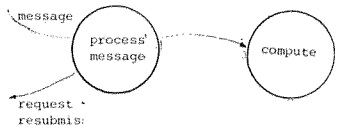


Figure 3(a). Retransmission of Single Message.

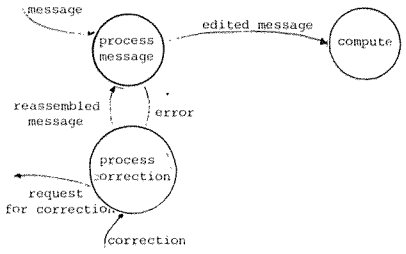


Figure 3(b). Single message transaction with separate correction message.

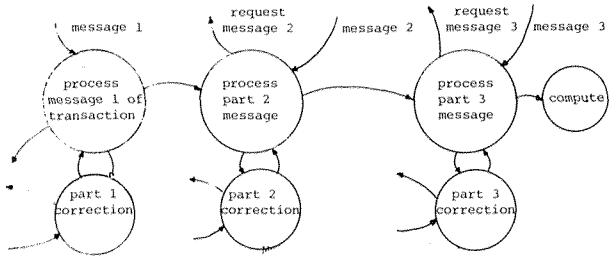


Figure 3(c). Multi-message translation with a different correction message for each stage.

Figure 3(d). Batched Transactions.

- store messages in computer memory and make them available to programs, or
- receive and transmit messages to terminal devices.

Many transactions processors also impose constraints on the programmer, as for example

- database record currency not maintained between module activations,
- each incoming message must be confined to one module,
- all database interactions confined to one module,
- using a common data area for the life of one transaction, or
- restrictions on the size of program modules.

Transaction Modes

In a transaction processing system, the unit of processing (at least from the user's point of view) is a transaction, and it seems logical that the program structure should reflect this. Transactions themselves call for a variety of processing modes. Figure 2 illustrates one mode only. In this mode each transaction input consists of one message. A message error requires the entire message to be input again. Other modes include

- single message transactions with corrections only resubmitted on error detection,
- multi-screen transactions where a transaction is built up in successive messages with local corrections at each step, or
- the batching of transactions for error control.

The fundamental structures for these three cases are illustrated in Figure 3 and described following.

A simple transaction, consisting of one input message, followed by some processing and generation of an output (e.g. a screen display) would be represented by a data-flow diagram something like that in Figure 3(a). Any error found in the message requests the repetition of the entire message. A variation is shown in Figure 3(b), where a correction screen is requested. The correction is received, the message is reassembled by replacing original fields by new fields and the message is edited again.

A typical multi-screen transaction involves entry of a sequence of messages, the response to each of which solicits the next, giving a data-flow diagram of the form shown in Figure 3(c). Here there are a number of stages. Each stage is similar to Figure 3(b) and solicits, edits and corrects one screen of the multi-screen transaction. The last alternative consists of batched transactions shown in Figure 3(d). In Figure 3(d) it is assumed that input is batched and consists of more than an unrelated collection of messages of the same type. More usually, a batch will consist of:

(i) a header screen, containing information relevant to the batch as a whole;
(ii) a series of detail screens; and
(iii) a trailer screen which includes control totals for the transactions in the batch.

The following paragraph describes batch processing in more detail.

Screens are entered in a set sequence. A sign-on (or acceptance of a previous batch) causes a blank header screen to be displayed. The operator fills this in, and sends it. If it is accepted, a blank detail screen is displayed. The transactions are then entered using any of the three earlier modes until a special end transaction is input. Acceptance of the end-transaction means that the batch is complete. Validation of the data entered occurs in two stages. Each transaction is checked in isolation immediately on entry. If any errors are found, the operator is asked to retransmit a correction. The second stages of validation occurs on reception of the trailer screen. The batch totals input on this screen are compared with those calculated from the transaction. If they agree, then the batch is accepted. Otherwise, the operator must locate the error and correct it. This could involve correcting the trailer screen, or one or more detail screens. It could also involve paging through detail screens to locate the error. Once the corrections are made, the operator indicates this, and the batch totals are again checked before acceptance of the batch. The significant difference between this error correction process and the original input operation is that the sequencing of screens is now under operator control. Facilities to assist this (numbering of detail screens, paging) are usually provided.

Structured Design and Transaction Processing

To be successful in the transaction processing environment, structured design must be adapted to transaction driven rather than program driven execution control. Furthermore, it must be applicable to all of the above transaction modes. Three possibilities of doing so are examined in this paper, namely,

- transaction control superimposition on the module structure chart,
- the adaptation of the data flow diagram to the input driven systems,
- an alternate reduction to that shown in Figure 2; this

reduction is fundamentally based on homologous structures.

The methods differ not so much in the module structures in the final design but in the reductions used to obtain these structures.

The first of the three methods uses the standard transformation of Figure 2 to generate the model structure. In the second case a transformation is first made on the data flow diagram and then the standard transformation is used. The third method takes the control structure directly from the data flows.

The three possible methods are now illustrated for the multi-message transaction mode illustrated in Figure 3(c).

Alternate Conversions

Conventional transform analysis would regard the data flow shown in Figure 3(c) as the afferent branch and represent it as shown in Figure 4. The full lines illustrate the flow of hierarchical controls. The processing would commence by successive calls to GET-MSG-1. Procedure GET-MSG-1 would obtain the message, validate it (by a call to VALIDATE-MSG-1) and request a correction if a validation error is found. The correction would be processed and used to reassemble the message. Control would then be returned to GET-MSG-2 where the process would be repeated for the second stage of the transaction. This will not work on a transaction processing system. In a transaction processing environment the system solicits a message. When message 1 of the transaction is received, the program is initiated at its entry point. In Figure 4 the entry point is MAIN-CONTROL. Once MAIN-CONTROL is activated control is passed to GET-MSG-1 which gets the message from its input buffer. To get MSG2, one must again solicit input (in this case message 2), which executes. When message 2 is received, the program is reinitiated at MAIN-CONTROL and control again passes to GET-MSG-1 rather than GET-MSG-2. Hence there is unnecessary flow through GET-MSG-3 leading to loss of functional cohesion in this module. Flow of control could, of course, be changed by use of flags, but this is messy and defeats the point of the whole structure design process, which is a provision of appropriate control structures.

ALTERNATIVE 1 – Transaction Controls

This alternative ignores the hierarchical control which is associated with structure charts. Instead transfers of control are effected by the transactions processor. In this case

- messages activate the modules, and
- modules activate other modules through the transactions processor rather than by subroutine calls.

These activations are illustrated in Figure 4. Now some of the hierarchical controls are replaced by transactions controls (shown by dotted lines). It is assumed that initially all modules are activated (by the transaction processor) and are in a wait state. Thus GET-MSG-1 is waiting for message 1 to arrive, GET-MSG-2 is waiting for message 2 and so on. How the particular message activates its corresponding module depends on the transaction processor. Possibilities are that

- each message has a header field to select a module, or
- a module indicates to the transaction processor the next module to be activated by an incoming message.

TABLE 1. Converting Pseudo-Code

```
GET-MSG1                                    GET-MSG1
    CALL GET-MSG1-FROM-BUFFER                   WAIT FOR MESSAGE1
    CALL VALIDATE-MSG1                          CALL GET-MSG1-FROM-BUFFER
    IF OK THEN send request for message2        CALL VALIDATE-MSG1
                RETURN                          IF OK THEN send request for message2
          ELSE CALL REQUEST-CORRECTION1                 SOLICIT GET-MSG2
               CALL PROCESS-CORRECTION1           ELSE CALL REQUEST-CORRECTION1
               RETURN                                  SOLICIT PROCESS-CORRECTION1

GET-MSG-FROM-BUFFER                         }
    moves message to working                }
    stores area                             }
    RETURN                                  }
                                            }
GET-MESSAGE-1                               }   remains the
    perform edit checks on fields           }
    IF NOT ERROR THEN RETURN OK             }   same
                SELSE RETURN error          }
                                            }
ASK-CORRECTION-1                            }
    send error diagnostic to user           }
    RETURN                                  }

PROCESS-CORRECTION-1                        PROCESS-CORRECTION-1
    CALL GET-CORRECTION1-FROM-BUFFER            WAIT FOR CORRECTION MESSAGE
    CALL ASSEMBLE-MESSAGE-1                     CALL GET-CORRECTION1-FROM-BUFFER
    RETURN                                      CALL ASSEMBLE-MESSAGE-1
                                                SOLICIT GET-MSG-2
GET-CORRECTION1-FROM-BUFFER                 }
    moves correction to working             }
    storage area                            }   remains the
    RETURN                                  }
                                            }
ASSEMBLE-MESSAGE-1                          }   same
    replace corrected fields in             }
    original message                        }
    RETURN                                  }
```

HIERARCHICAL CONTROL — VIRTUAL CONTROL

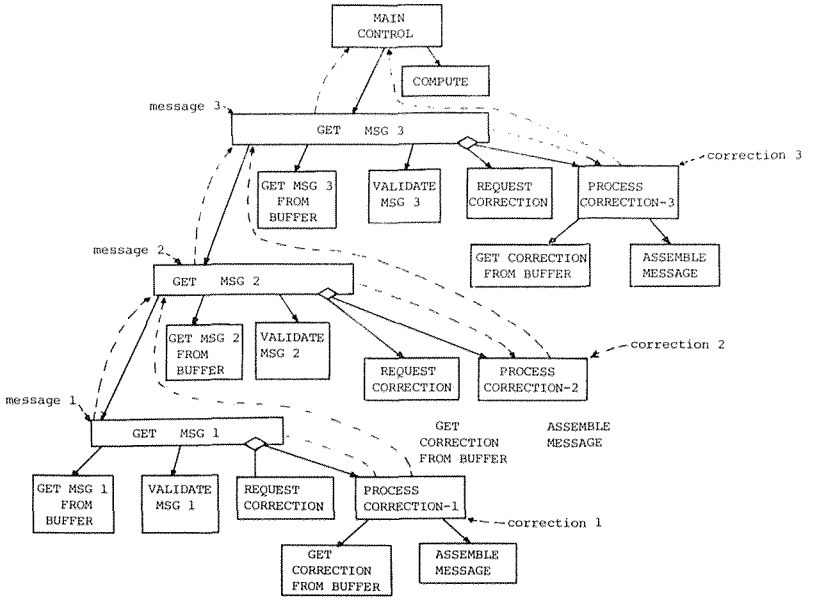


Figure 4. Converting Hierarchical to Virtual Control.

In Figure 4, message 1 activates GET-MSG-1 which then calls GET-MSG-1-FROM-BUFFER (to get the message from its buffer and VALIDATE-MSG-1 to edit the message). It will then solicit either message 2 (indicated by the dotted line to GET-MSG-2) or correction (indicated by the dotted line to PROCESS CORRECTION-1). Subsequently GET-MSG-2 is activated. From a designer's viewpoint, design would commence using structured design techniques to develop a hierarchical structure chart. The difference from structured design occurs when modules are designed and pseudo-code is developed. The pseudo-code must now contain the commands to implement the transaction processor controls rather than the hierarchical controls. The difference is illustrated in Table 1. Now some returns have been replaced by SOLICIT statements. These indicate the module to be activated by the next incoming message. For example, SOLICIT GET-MSG-2 in module GET-MSG-1 indicates that the next message (namely message 2) is to be processed by GET-MSG-2. For clarity, a WAIT statement has also been used in Table 1. The WAIT statement at the beginning of a module indicates the message that will activate the model. The WAIT statement need not usually be included in the code as this information is usually maintained by the transaction processor.

ALTERNATIVE 2 – Adapting the data flow diagram to transaction processing

It can be argued that the reason why the transformation from data-flow diagram to program structure does not work is that it is the wrong data flow diagram. All the data flow diagrams (DFDs) in Figure 3 are logical ones representing the user's view of the system, rather than physical ones that represent its implementation. For a conventional system, the difference between the logical and physical DFD's are often trivial, so one usually does not bother to draw the physical DFD, merely indicating physical elements (e.g. the person-machine interface) on the logical DFD. To draw the physical transaction processing DFD one must remember what happens in a transaction processing system when a solicit/response sequence occurs; a prompt is sent to the terminal, the data area for the transaction is stored, and the program terminated; on receipt of the input message, the data area is recovered, and the load unit designated to receive the message is initiated. The physical DFD corresponding to Figure 3(c), for example, is shown in Figure 5. The three messages correspond to entirely disjoint processes, which interface only via data stores and the external entity (the terminal operator), not via any data flow, and as such should be represented by entirely separate program structures, as shown in Figure 6. In this figure, GET, VET and SOLICIT are the afferent, transform and efferent branches respectively of the MSG1 and MSG2 handling programs. Message 3 is handled in the afferent branch of a program which then processes the transaction. Solicit/response transfers of control are shown as dotted arrows. These are not derivable from transform analysis.

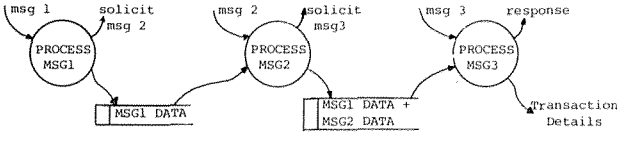


Figure 5. Physical DFD.

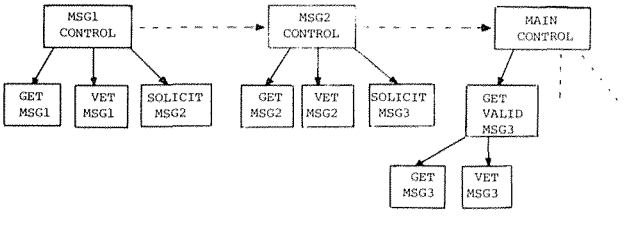


Figure 6.

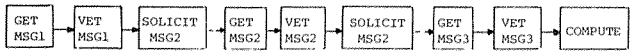


Figure 7. Homologous Converstion.

ALTERNATIVE 3 – Using a homologous transformation

The homologous transformation converts the data flow diagram directly to modules. The flow of control corresponds to the flow of data between data flow diagram processes. An example of a homologous conversion for the multi-message transaction mode is shown in Figure 7. The homologous structure provides almost no discipline in formulating programme structure, resulting in potentially highly interconnected (and spaghetti-like) structures.

EFFECTS OF OTHER TRANSACTION PROCESSOR CONSTRAINTS

Apart from control structure, transaction processors can include other constraints, which can prevent satisfactory coupling and cohesion. Ways of dealing with two of them namely, common environment coupling and load module restrictions, are discussed following.

Common Environment Coupling

Transaction processing systems normally have a work area set aside for each transaction, which is stored between messages and so may be used to pass information between transaction steps. It is possible to use this area in either of two ways.

(i) As a common data area for all modules

In this case, the layout of the area is defined once, and copied into all load units. Information in the area may be accessed or altered by any load unit. This results in the data inputs to modules not being well-defined, and can result in corruption of data in a way which is hard to locate. This type of organisation, in which all modules are coupled by a common data environment, is undesirable and should be avoided.

(ii) As a parameter area

In this case, the area is used specifically for the passing of parameters between load units, and so is different for each call. There is a difference between this and a hierarchical call, however in that in the hierarchical call, input and output parameters are passed to and from the same calling module, while in a homologous structure, the calling and called modules are different. Thus in the hierarchical call, it is possible to allow input and output parameters to appear in the same parameter area:

```
01 PARAMATER-AREA
  03 INPUT-PARAMETERS
       .
       .
       .
  03 OUTPUT-PARAMETERS
       .
       .
       .
```

In the homologous call, it is important that the called module should not know what the calling module passed to the current module (unless we specifically want to tell it), so we must suppress the input parameter area. The best way to do this is by overwriting it:

```
01 PARAMETER-AREA
  03 INPUT-PARAMETERS
       .
       .
       .
  03 OUTPUT-PARAMETERS REDEFINES INPUT-
     PARAMETERS
       .
       .
       .
```

Hence each module passes a block of parameters starting at the beginning of the work area.

Many transaction processing systems allow a load unit to discover which load unit called it, so that a load unit called by more than one other unit could, in theory, be passed different sets of input parameters, depending on the caller. This seems, however, to be an unnecessary complication, and at variance with the practice in calling subordinate modules, where the same parameter set (even if some parameters are null) is always passed.

Effect of load unit size restrictions

In many transaction processing systems, the load unit size is restricted, since they are treated as overlays which are loaded into an area of restricted size. This may mean that the program processing a particular message may need to be divided between two or more load units. This involves replacing a hierarchical call with a direct transfer of control, from which there is no return. To do this, one must identify places in the structure where the control characteristics correspond to this requirement. A common case is the junction between afferent and transform and transform and afferent branches, e.g. it is likely that the structure in Figure 2 could be decomposed to give that in Figure 8(a). Another situation is when a transform involves a series of consecutive steps (Figure 8(b)). In this case the hierarchical structure imposed by transform control is simply eliminated (Figure 8(c)). In performing such transformations, the major object is to avoid introduction of loops around groups of load units, or complex patterns of control transfer.

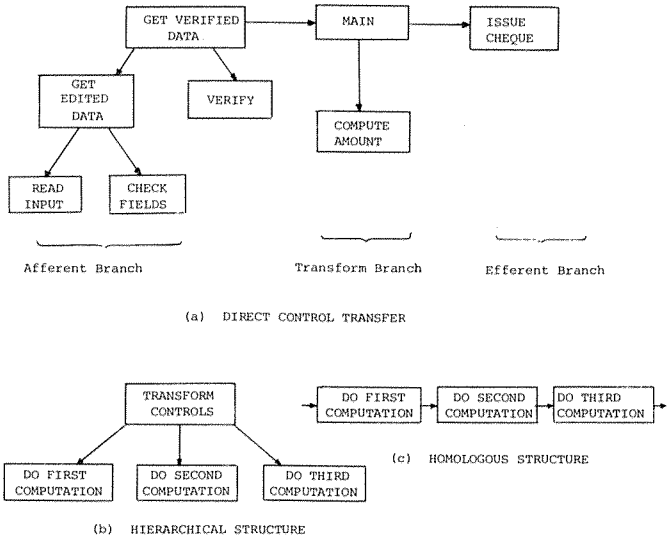


Figure 8. Alternate Reductions.

CONCLUSION

The paper has discussed the problems that arise when hierarchical reductions are used in the transaction processing environment. The problems arise because in this kind of environment it is the transaction processor and not the programs that effects transfers of control. The paper then outlined alternative methods that use the principles of hierarchical reduction but still allow the transaction processor to exercise transfers of control.

REFERENCES

CLARKE, R.(1982): Teleprocessing Monitors and Program Structures, *Austral. Comput. J.*, 14, No. 4, November 1982, pp. 143-149.

DeMARCO, T. (1978): *Structured Analysis and System Specification*, Yourdon Press.

GANE, C. and SARSON, T. (1978): *Structured Systems Analysis*, Prentice-Hall.

YOURDON, E. and CONSTANTINE, L. (1979): *Structured Design*, Prentice-Hall.

BIOGRAPHICAL NOTE

David Walker obtained his PhD in Theoretical Physics from Melbourne University in 1970. Since then he has worked as a research scientist and in the Australian Public Service, and is now a lecturer in Information Systems at Canberra College of Advanced Education.

Igor Hawryszkiewycz received his BE and ME degree in Electrical Engineering from the University of Adelaide and his PhD in Computer Science from the Massachusetts Institute of Technology in 1973. Currently he is the Principal Lecturer in Information Systems at the Canberra College of Advanced Education and before that worked at the Research Laboratories and subsequently the Data Processing Branch of the PMG Department (now Telecom). His main interests are system design and database systems.

SHORT COMMUNICATION

A System for Visible Execution of Pascal Programs

By R. F. Hille* and T. F. Higginbottom†

This paper describes a system for the visible execution of Pascal programs, intended for use in computing laboratory classes for first year students. It is essentially a pre-processor, written itself in Pascal for portability. It inserts additional code into the user program to cause the display on the terminal screen of the current statement together with its line number in the source as well as the value of any variable that was changed as a consequence of executing that statement. Break points are set to enable the user to step through the Pascal code by executing one statement at a time. The pre-processor is imbedded in a UNIX shell file to bind the whole system into a package.

Keywords and phrases: Pascal, preprocessor, visible program execution.
CR Categories: D.3.3, D.3.4, K.3.2.

1. MOTIVATION

First year students of computing science at this university learn to program in the teaching language Pascal, using the department's Perkin-Elmer 3230, running under version 7 of the UNIX** System. It has become clear to us that beginning students often have problems understanding the sequential nature and dynamic aspects of their programs and many programming errors result from that lack of understanding. Therefore, in the past we usually encouraged them to insert additional write statements for two purposes: to indicate in the output in which part of the program the action took place, and to give the current values of critical variables.

Such a procedure has some drawbacks: it is quite laborious to insert all the necessary write statements, it becomes necessary to replace some single statements by blocks enclosed in begin and end, the action rolls off too quickly, finally, the program loses its readability due to the addition of a substantial number of statements. We decided to automate the entire process and develop a pre-processor which inserts the additional code. Furthermore, the user must be able to step through the program at his own pace. That requires the setting of break points.

A number of other systems with different aims have been developed in recent times. Hodgson and Porter (1980) describe a system for a language of their own design, which allows execution of programs at varying speed and is capable of simulating execution in reverse time by keeping the complete history on file. Barter and Hodson (1981) have developed a screen based interpretive Pascal system with some additions to standard Pascal, which consists of an interactive interpreter for the compiled intermediate code with trace-back to the Pascal-source. The difference between their system and ours is that we operate entirely at the source code level, in the interest of ease of implementation. It took one of us two weeks to design, implement and debug the entire system. This was possible because our system is simply a pre-processor. Teitelbaum and Reps (1981) have developed an interpreter for source code of PL/I, which runs on an LSI-11 based micro computer. It allows the user to change the source code interactively while keeping the symbol table up to date so that program execution can be resumed at the point where it was interrupted.

The intention of our system is to demonstrate to the student user the execution sequence of his program in order to help him understand its dynamic properties. We believe that finding a programming error, and altering the design of a program to rectify the fault must be two separate actions. Changes "on the fly" are dangerous because they can be very time consuming and may lead to unsatisfactory results. They also encourage bad programming habits.

2. DESIGN AND IMPLEMENTATION

Our system accepts standard Pascal (Jensen and Wirth, 1975), (Joy, Graham and Haley, 1977), except for some restrictions which we plan to remove in future versions. The Pascal statements of the source code must be on separate lines. For example, the guarded statement

if <condition> then <statement>;

must be written as

if <condition> then
<statement>;

This restriction is not as severe as it may seem, because our students already write their programs that way. We encourage students to write readable programs and to use indentation to identify the block structure of their programs. They are encouraged to write only one statement per line and to keep the keywords *begin* and *end* on separate lines.

Source statements are displayed by inserting them together with their line numbers as strings into writeln state-

**Unix is a trademark of Bell Laboratories.

*Department of Computing Science, University of Wollongong, Wollongong, NSW 2500. †Department of Computer Science, Southeastern Louisiana University, Hammond, Louisiana 70402, USA. Manuscript received January 1982, revised September 1982.

ments. Other *writeln* statements for additional information are assembled in similar fashion. Break points are set by inserting *readln* statements. This may cause problems with tests for end of line in *while* loops, because the test cannot be decided for standard input unless reading actually takes place. We solved the problem by re-arranging the sequence of events in some places. Unfortunately we cannot display the current values of set variables, because Pascal cannot print sets on standard output.

The user may want to step through only part of the program. Therefore, we implemented the commands

```
trace on
trace off
```

which can be placed as additional lines of comment around that portion of the program whose execution is to be displayed. The command "trace on" causes our display system to insert the required additional code until either the command "trace off" is encountered or the user program ends.

A number of considerations have led us to implement the system as a UNIX shell program. The name of the user program can then be supplied as an argument, which is not possible with Pascal programs. Error checking and printing of instructions to the user can be done by the shell program.

3. FUTURE DEVELOPMENT

The implementation as it exists at present has some limitations. The restriction on the input format will be removed by including a lexical scan. Maintenance of a partial symbol table will enable recognition of cases in which special action has to be taken for the display of assignments.

Stepping through loops may be time consuming and quite unnecessary. The user can be given the option to specify at break points, for how many steps he wants to suppress the display mode.

Output sequences from the modified user program preceded by a special character can be passed through a filter which interprets them as commands for a screen driver, so that a window into the source file can be displayed instead of just the current line.

The work to implement these extensions is in progress.

REFERENCES

BARTER, C.J. and HODSON, D.A. (1981): A Screen based Pascal System, *Austral. Comput. Sci. Commun.*, 3, p. 14-24.

HODGSON, L.I. and PORTER, M. BIDOPS: a bidirectional Programming System, *Austral. Comput. Sci. Commun.*, 2, p. 349-355.

JENSEN, K. and WIRTH, N. (1975): *PASCAL User Manual and Report*, Springer-Verlag, New York, Heidelberg, Berlin.

JOY, W.N., GRAHAM, S.L. and HALEY, C.B. (1977): *UNIX Pascal User's Manual*, Version 1.0., University of California, Berkeley, CA.

TEITELBAUM, T. and REPS, R. (1981): The Cornell Program Synthesizer: a Syntax directed Programming Environment, *Commun. ACM*, 24, p. 563-573.

Letters to the Editor

COMMENTS ON THE PAPER BY LAWRENCE AND JEFFREY

In their paper "Commercial Programming Productivity – an Empirical Look at Intuition" (*Australian Computer Journal*, 15, pp. 28-32, 1983), Lawrence and Jeffery conclude *inter alia* that structured programming has no significant impact on programmer productivity. Possibly they are correct; but I doubt that the evidence presented is sufficient to support this conclusion. A more likely conclusion, on the evidence I have seen, is that structured programming is not being used.

The program is one of definition. What is structured programming? More importantly, who decides whether a given program is well-structured? Although the authors do not say so explicitly, it seems likely that the description "structured programming" was supplied by the programmers or organisations studied, rather than by some independent assessor. It is interesting to note that Table 2 of the paper claims that a surprisingly high proportion of the COBOL programs studied were developed using structured programming techniques. I find that hard to believe. I have yet to see a well-structured COBOL program, except for toy examples; the language simply does not have the right features to support a clean programming style. My own experience has been that it is difficult to teach good programming habits to anyone contaminated by prior exposure to languages such as BASIC, COBOL and FORTRAN. The "line-at-a-time" philosophy has become too deeply entrenched.

A question I would like to see addressed in some future study is the effect of data structuring and name management in programs. I believe that these factors would strongly influence programmer productivity. In particular, I have long asserted that the use of variables whose name scope is too large – more than about half a page of code, as a rough estimate – will lead to programs which are hard to write and hard to maintain. This implies that, in languages like Pascal or PL/1, global variables should be avoided wherever possible, and that nested procedure definitions should be used with caution (if at all). It also implies that BASIC and COBOL should only be used for very short programs, because in those languages the scope of names is the entire program. Long programs in those languages are almost invariably hard to read.

This letter is not a criticism of Lawrence and Jeffrey. They did their best with the available data. However, I am inclined to believe that "structured programming" will remain a meaningless catch-phrase, having little effect on productivity, until such time when a significant number of organisations start using languages and support tools which properly support clean coding practices.

We have made a start on educating programmers. We have yet to reach those who make the decisions on what languages and system software will be used.

P.J. Moylan,
Department of Electrical and Computer Engineering,
University of Newcastle, NSW 2308

I found the paper "Commercial Programming Productivity – an Empirical Look at Intuition" by M.J. Lawrence and D.R. Jeffery most informative. However, I believe that one paragraph is worthy of further analysis, and I quote:–

"The majority language in the sample (216 programs) was COBOL with the remainder PL/1 and BASIC. Assembly programs were excluded from the sample due to the problems they raise with the productivity metric."

This paragraph gives rise, I believe, to a number of questions, including the following:–

- Does the exclusion of Assembly programs of itself invalidate the metric?
- Does this exclusion mean that a real difference in productivity exists between programs *developed* using assembly languages and those using high level languages?
- Could it be that in a study taking an empirical look at intuition, a valid sample has been excluded by intuition?

D. Evans, BSc, MACS,
Berger, Jenson & Nicholson (Aust.) Pty Ltd,
Rhodes, NSW 2138

AUTHORS' REPLY

We appreciate the time taken by both P.J. Moylan and D. Evans in commenting on our paper "Commercial Programming Productivity – an Empirical Look at Intuition".

Moylan questions the definition used for structured programming and whether COBOL programs can ever be structured. We used as our definition the text "Elementary Structured Cobol" by Davis, *et al.* (1977).

While our research was limited particularly to COBOL programs, evidence available from other studies suggests that similar findings relating to structured programming would have resulted if other languages had been used. Sheil (1981) and Vessey and Weber (1982) reviewed and evaluated the literature relating to empirical studies on structured programming. These studies examined such structured programming issues as indenting, control flow, variable naming conventions, commenting and maintenance. No clear conclusion emerges from these studies as the observed effects are usually weak and generally not statistically significant. Thus our study agrees with prior empirical studies.

With regard to the questions raised by Evans, we believe the exclusion of Assembly programs may reduce the scope of application of the findings to COBOL and other third generation languages, but does not invalidate the productivity metric used.

M.J. Lawrence,
University of New South Wales,
Kensington, NSW 2033

REFERENCES

DAVIS, G.C., OLSON, M.H. and LITECKY, C.R. (1977): *Elementary Structured COBOL: A step-by-step Approach*", McGraw-Hill, New York.

SHEIL, B.A. (1981): The Psychological Study of Programming, *ACM Computing Surveys*, 13, pp. 101-120.

VESSEY, I. and WEBER, R. (1982): Research on Structured Programming: an Empiricist's Evaluation, Department of Commerce, University of Queensland (draft paper).

Book Reviews

DEMPSTER, M.A.H., LENSTRA, J.K. and RINNOOY KAN, A.H.G. (ed.) (1982): *Deterministic and Stochastic Scheduling,* (Nato Advanced Study Institute Series), D. Reidel Publishing Company, Dordrecht, Holland, 419 pp. $US48.00.

In recent years, it has become common to incorporate in a technical conference some sessions which deal not with individual research reports but with tutorials or survey lectures on the state of the art. This development is helpful, not only for those on the fringe of a specialist discipline, but also for those already active in it.

In July 1981, a NATO Advanced Study and Research Institute on Theoretical Approaches to Scheduling Problems was held in Durham, England. At this Institute, eight of the ten days were devoted to study, and two to research. The volume under review consists of the proceedings of this Institute, and in comparison with the average conference proceedings, benefits greatly from the unusual balance.

The fifteen contributions to the first part of the proceedings provide an extensive survey of theoretical scheduling, both deterministic and stochastic. The 39-page survey of deterministic scheduling by Lawler, Lenstra and Rinooy Kan is outstanding and should be required reading for anyone interested in this field. The latter two authors, both Dutch, have been active in the field for a number of years, and have demonstrated an impressive grasp of the extensive literature in scheduling theory.

The nine contributions to the second part of the proceedings are the usual mixed bag of current research reports. The book is well indexed and highly recommended to people working in the fields of operations research or computer science who have an interest in scheduling problems.

G.B. McMahon,
University of New South Wales

SCHNEIDER, H. and WASSERMAN, A.I. (eds.) (1982): *Automated Tools for Information Systems Design.* Proceedings of IFIP WG 8.1 Working Conference, North-Holland, Amsterdam, 261 pp. $US39.50.

IFIP Technical Committee 8 and its working groups have been sponsoring some very interesting conferences recently in the systems design area. This book publishes the papers of one such conference. The papers presented provide a good cross-section of the type of work being carried out, including:

- graphics, for example, for the support of structured analysis and for the visualisation of program execution,
- grammars and on-line tools for schema definition,
- evaluation and monitoring systems,
- grammars for system definition, and
- on-line design-support systems.

Many of the tools are in their early stages as yet. Therefore the professional system designers will not find a product to solve too many of the day-to-day problems, but they will find many approaches to the support of designers that will help to consolidate their ideas or suggest improvements that could be made in methodologies in use, or approaches to the design workforce.

There is no doubt that this is a very significant area in systems research and product development, and that these types of conferences provide a most up-to-date perspective on developments. Therefore, the person who has an interest in design methodologies or tools for design will find much of interest in this book. The papers are generally very readable and suggest many interesting possibilities for computer support for the system designer and builder.

D.R. Jeffery,
University of New South Wales

TROPPER, C. (1981): *Local Network Technologies,* Academic Press, 144 pp. $US19.95.

Local computer networks have begun to appear in the "Automated Office", their design and performance is actively studied by major computing research groups and they are the subject of a major standardisation effort. It is not surprising, therefore, that books on the subject should begin to appear. From the title the reader of this book would expect it to be wide ranging and to cover many aspects of Local Computer Networks. The author describes the book in the preface:

"The purpose of *Local Computer Network Technologies* is to synthesize the considerable amount of work accomplished in developing link access protocols for ring and bus computer-communication networks and to provide a systematic discussion of both the protocols and their associated performance models."

The book is divided into two major parts: one describing Ring networks and one describing Bus networks. Each part contains descriptions of network access techniques, referring to particular implementations, followed by a detailed analysis of the performance of the technique.

The coverage of link access procedures is reasonably complete though there are some notable omissions when particular implementations are described. For example the Xerox Ethernet is dismissed in one sentence and the Cambridge Ring is not mentioned at all! The particular techniques used by these systems are described but the book would be of little use for someone without knowledge of the local network area that wanted to study the performance of the Cambridge Ring for example.

In summary I would not recommend the book to someone with a general computing background that wanted to learn about local computer networks. It is of use mainly to the specialist in the area who requires a reference text on the performance of the link access procedures used in local networks.

R. Kummerfeld,
University of Sydney

MARTIN, J. (1982): *Application Development Without Programmers,* Prentice-Hall, 350 pp. $A50.50.

I was disappointed with this book. That comment must, however, be counter-balanced by another: that it is one of the most important books ever published in the area of commercial application development.

Martin's strengths have always been in his wide technical scope, his fluent delivery and his ability to present simple explanations of potentially very difficult ideas – he is in short the populariser par excellence. This book's weakness is its lack of structure: it meanders repetitively, quite apparently developed on the basis of a variety of seminars and failing to integrate the diverse materials.

Because those seminars were addressed to a variety of different clients, the book suffers too from uncertainty as to its target audience. Is this clumsiness a sign that Martin's becoming bored with books and more interested in the burgeoning audio-visual and computer-assisted communication environments to which his style is so well suited? This is after all his twenty-fourth book for Prentice-Hall alone.

I review it as I think it should have been – three separate and much shorter books. The first of these 'books' is addressed to *users,* with the title "Application Development Need Be a Mystery No Longer!" It comprises sections from the first 15 chapters and 242 pages of the real book, but requires considerable re-working. The problem is that, although ostensibly written for end-users, these sections contain a great deal of detailed argument of the kind needed to convince senior data processing professionals that conventional detailed design and coding can be automated.

Highlights of this 'book' are the discussions of the various classes of inquiry and reporting tools (pp. 14-25) and the characterisation of the wall between user and programmer (pp. 77-80). There is also a very helpful section on data base user languages (pp. 118-147) and as simple an introduction to APL as a non-technical person would ever find (pp. 189-198). On the whole though the DP-literate end-user is less well served than he might have expected from a book classified by the publisher as 'suitable for end-users'. The material is inadequately structured and unclearly sequenced, and a naive end-user would find himself buried in much (to him) irrelevant material.

The second 'book' is for *DP Professionals* and argues that the desperately-needed increase in software development productivity cannot be achieved by the third-generation, essentially procedure-oriented, methodologies and tools. A multi-facetted revolution is already underway, involving the creation of stable data models as a basis for quick development of applications via predominantly non-procedural specifications.

The central role of data base is continually stressed, though I have the feeling that even Martin has difficulty at times unravelling the unfortunate confusions between the ('logical') definition of data elements and their relationships and the functions of physical data storage and retrieval.

A poor chapter examines Performance Considerations, suggesting that performance degradation is nothing to worry about. Elsewhere he is far more prudent, pointing out that different products show wildly different comparisons with an equivalent COBOL program due to the wide range of possible implementation strategies.

The final 43 pages discuss organisational implications. Most of it is an unsatisfying re-hash of the 'Information Center' cluster of ideas. This phrase (in no sense is it a coherent construct) is used to refer to virtually any organosational arrangement different from the one or two conventional heirarchies. The chapter on the changing role of the systems analyst is better, though it continues to confuse the ideas of system design and system analysis.

Like the user, the DP specialist is left somewhat dissatisfied with his 'book'. The new method of development isn't clearly painted, though it seems to involve less procedural coding and fewer machine-oriented formalisms (like having to first define work-areas, and using clumsy DBMS interfaces). It would have helped if the discussions of 'non-procedural' and 'fourth generation' (pp. 26 and 28) hadn't been so facile. In his current seminar-series Martin has at least improved his explanation of the former by instancing instructions to a taxi-driver: 'go 300 metres, turn left . . .' is a set of procedural instructions; 'take me to the Odeon' or 'take me to "Star Wars"' is a non-procedural because it delegates the question of 'how' to the supporting facilities. He still offered no better explanation of 'fourth generation' than 'the one after the third . . .'

Specialists in the area will be concerned about a range of other deficiencies, which for brevity's sake I list in point form:

- the vital questions of software portability and 'lock-in' to existing hardware and software suppliers are barely mentioned;
- there is a heavy orientation towards IBM, reflecting as much Martin's associations as that company's contributions to the field;
- it dismisses preprocessors without adequate discussion, or even, apparently, thought;
- it is insufficiently critical of the deficiencies of existing products (with the exception of ADF on pp. 258-263, which seems like a supplier-approved token sacrifice) and accordingly is unable to suggest how to avoid the pitfalls;
- it blithely ignores the common sources of error in third-generation programming (like tortuous logic resulting from poor data-structuring, and incompatible data-types) and thereby condemns those mistakes to be repeated in the fourth generation (in a more powerful manner of course);
- it glosses over the difficult area of data base design (which is, admittedly, well-treated in others of Martin's publications), and in particular omits discussion of the automation of data base design (as distinct from automatic navigation through an existing structure). Some of the fourth generation products already have this feature;
- it omits from the main table on pp. 20-24 at least three important products available in Australia (Delta, LINC and Natural), reflecting the book's heavy US bias. It also expresses inconsistent opinions about the suitability of FOCUS and RAMIS II for end-users and for DP professionals (on pp. 19, 23 and 309);
- it discusses prototyping (pp. 64-65) in a marketing rather than a critical manner, failing completely to differentiate between a designed experiment and trial-and-error.

The third 'book' within a book is, or should have been, a more rigorous analysis of the deficiencies of existing methodologies and tools, of the environment of application development and of the identifying characteristics of the alternative culture. The target for such a book would clearly be the *applied computing scientist.*

Such discussion as is provided (mainly in Chapters 1, 4 and 6) is assertive, with inadequate reference to the literature. While it rings true to myself (a DP specialist), the argument as presented could at best be awarded a 'case unproven' verdict. Another source of concern is the non-disjunct classification schemes Martin continually uses: they're good for overcoming ignorance, and can be valuable communication devices even for experienced professionals; but they turn to water when you analyse them or try to use them as a basis for prediction.

A few other quibbles:

- it hardly feels satisfactory to have such a vaguely-justified title. "Application Development Without Conventional Coding" would be more accurate;
- Figures and boxes are arbitrarily differentiated in the text, making it unnecessarily difficult to find an exhibit (on the other hand the frequent, useful checklists are one of the hallmarks of Martin's books);
- the text contains more errors than are normal in such a publication (list supplied to publisher). It's hardly been rushed off the press, having been written in 1980, published by the Savant Institute in 1981, then by Prentice-Hall in the second quarter of 1982 (reaching Australia only in the fourth quarter);
- of the 115 references appearing at the end of chapters, 35 are to IBM or IBM-related publications and a (record-breaking?) 26 to Martin's own.

So, with that catalogue of criticisms, why is the book so important?? Quite simply, because it fills a hitherto gaping hole in both the academic and the practical literature. Learned journals are publishing (at times quite petty) arguments about late third generation languages like Ada and abusing practitioners for not adopting software engineering practices. Meanwhile a wide range of pragmatically developed tools have appeared with the potential to render programmers' work-benches, work-styles and education courses largely irrelevant in a matter of a few years. Apart from the most highly competitive embedded systems (mainly weaponry), Martin's own guess is for the death of new third generation language programming by 1990.

There is much in the book that is poorly organised, there are omissions, and the implications of this new technology are far from satisfactorily explored, but Martin has provided practitioners with a base to work from. It is a matter for conjecture whether developments in this field will be dictated by software suppliers and practitioners alone, or whether the academic world will commence making contributions.

Roger Clarke,
Xamax Consultancy Pty Ltd, Leichhardt 2040

LEWIS, R. and TAGG, E.D. (eds.) (1982): *Involving Micros in Education,* North-Holland, Amsterdam, 248 pp. $39.50.

This is the proceedings of a conference held at the University of Lancaster, England, in March 1982. In order to achieve publication within a year, original material has been taken and photo-reduced, so the reader should approach his task with a magnifying glass and a tolerant view of typographical errors.

There are 37 papers, mainly of the 'this is what we did and how we did it' type. The papers are grouped under 12 headings, which are presumably the stream titles from the conference. However, the headings are merely indicative of the topics covered by the corresponding papers and do not accurately reflect the contents. In particular, some of the distinctions made are unnecessary; for example, most of the papers on the sections headed 'Micros in Science Laboratories (1)', 'Micros in Science Laboratories (2)', 'Applying Micros to Courses', 'Microprocessor Courses', and 'Microprocessors in Control' could have been gathered, together with some papers from other sections, under the single title 'Microprocessors in Measurement and Control'. The two major emphases in this group of papers are: teaching about computers with a measurement or control application as practical work; and the microcomputer as a tool for measurement and control in the science or music laboratory. In addition to these papers, there are others which describe particular laboratory or network configuration of microprocessors which have been established.

Some of the other papers are disappointingly ordinary. In J. Hebenstreit's opening address, for example, he introduces the terms 'teachware' for primary CAI and 'courseware' for adjunct CAI, but otherwise just re-runs the 'Wired City' scenario. E.T. Powner confuses education *with* and *about* computers; another paper describes graph-plotting as if it were a novelty; and another describes the use of keywords in the same vein.

The most interesting paper is probably R.S. McLean's 'The Microcomputer has a Role in Human Evolution', which develops Papert's ideas, and sets a definition of computer literacy in a theoretical context. Other useful papers are by W.Y. Arms, exploring the merits of a mixture of personal and time-sharing computers; by J.B.H. Du Boulay, J.A.M. Howe and K. Johnson, who discuss the extent to which learning programming is necessary in mathematics education; by J. Short, R. Brown, A. Kilgour and M. Dodain, who describe microcomputer assistance with essay writing, and also judge Pascal to be a better author language than Pilot; by N. Parezanovic, defining an author language with more of the characteristics of a normal programming language; by M.B. Foulis, giving a useful check-list for the evaluation of CAI materials; and by D. Watson describing the benefits of combining programmers and teachers into a writing team for best results when writing CAI material.

In summary, this book contains some papers of interest to

anyone concerned with the development of computer use in education; and a lot of papers, any one of which may offer just the right assistance to someone pondering a possible control or measurement application of a microcomputer.

D. Woodhouse,
La Trobe University

BENNETT, J.H. and KALMAN, R.E. (eds.) (1981): *Computers in Developing Nations,* North-Holland, Amsterdam, 272 pp. $US36.25.

This book contains reprints of papers presented at the Seminar on Computers in Developing Nations, organized by TC-9 of IFIP. It was held in Melbourne on Monday, 13 October 1980, immediately prior to the Australian segment of IFIP-80.

The pages are neither typeset nor edited. They are simply reprinted from camera ready copies of papers as submitted by the authors. The price of $US36.25 for 272 pages must be some sort of record. Each paper spends at least a page on a description of the importance of the computer revolution to mankind and society. This increases the soporific value of the book considerably. I could manage no more than three pages on most nights. The book would be greatly improved by stringent editing of all repetitive introductions.

Two main themes emerge from the papers. The first theme recognizes the shortage of skilled manpower and the lack of communication equipment in developing countries and describes a variety of approaches of how the governments of these nations are trying to overcome these handicaps. The second theme states with great unanimity between several speakers that a central plan is essential for the development of computer applications.

In the name of efficiency such plans propose to restrict equipment purchases to one or two manufacturers, restrict software development to one or two laboratories, restrict computer power to a few large computer centres and restrict system analysis to a few chosen institutes. All these suggestions are made in good faith in the name of efficiency and economy but in practice they stifle initiative and foster waste and corruption.

In summary, the papers provide a good overview of the problem facing computerization in the developing countries and how the governments of these countries plan to overcome such problems.

Juris Reinfelds,
University of Wollongong

BRITISH COMPUTER SOCIETY (1981): *Control and Audit of Minicomputer Systems,* Heyden & Son, London. 52 pp., $16.00.

This review paper comprising 52 pages has been produced by a working party of the British Computer Society 'Auditing by Computer' Specialist Group. It is not a complete guide to controls for minicomputers but concentrates on particular differences characteristic of minicomputer environments. This decision by the working party has resulted in a concise, readable text, but makes it difficult for any reviewer to know whether any omissions were intended due to their presumed applicability to both large and small system environments.

The appeal of the booklet lies in the logical layout of the subject matter, including a useful chapter on 'Controlling a Corporate Minicomputer Policy', and the concise simplicity of the approach taken by the authors. Practising auditors with limited experience of auditing EDP systems and new entrants to the audit profession with a basic training in system controls and EDP audit (e.g. having reached a standard equivalent to the Canadian course – CICAI and CICAII) will derive considerable confidence and reassurance from the conclusions reached by the working party. For example, the chapter on Word Processing Systems offers a practical guide to the control and audit requirements of commercial text processing operations. The practical value of the advice tendered in this paper is also enhanced by chapter 10 – New Areas for Auditors, by a very useful appendix of suggested further readings and by valuable insights into Governmental requirements. In relation to the last point the emphasis on value-added-tax (VAT) is not, of course, relevant in Australia, but the general concern of Governments to establish systems assurance requirements for applications processing customs and excise duties will strike a responsive chord! The booklet is essentially a primer for inexperienced auditors, is relatively expensive at $A16.00, but may well be considered a prudent personal investment by computer auditors lacking exposure to small business sytems.

More experienced auditors, however, are likely to share my opinion that the approach selected suffers from over-simplification and serious omissions and inadequacies. The material offered and conclusions drawn appear more suited to the mid-to late seventies, and the very real problems of auditing online minicomputer systems or coming to grips with the variety of distributed processing configurations now encountered are not addressed. More seriously, the need to focus on the auditability of online systems and the difficulty of establishing adequate audit trails in modern systems is ignored. The application of integrated test facilities, embedded data collection techniques or use of other concurrent audit techniques are apparently not considered worthy of mention. Even the need to design specific internal control questionnaires for small system environments fails to attract comment. Thus, readers with a particular interest in the auditability of small systems will find little to excite their attention. The chapter on 'Influence of Computer Hardware on Control' leaves a lot to be desired, and the frequent failure to clearly distinguish between control objectives and control techniques is confusing and disappointing.

In summary, the objective of the booklet of 'providing advice and views in several key areas rather than to provide a rigorous analysis of the whole topic' can fairly be claimed to have been met for auditors with limited experience of small systems. However, there is a wide gap between the limited objectives of the working party and the requirement for expert advice on modern practice in auditing the diverse minicomputer networks and sophisticated (online) small business systems now being encountered in increasing numbers by auditors. The booklet is disappointing in not exposing the more complex issues and serious problems now evident to many EDP auditors.

B.J. Garner,
Deakin University

CARTER, L.R. (1982): *An Analysis of Pascal Programs,* UMI Research Press, Ann Arbor, Michigan, 199 pp. $US56.50. Distributed by Bowker Publishing Company, Epping, CM164BU, England.

This book will have a restricted readership due to price and a very narrow aim, yet to the best of my knowledge its contents are not repeated elsewhere. It contains 27 statistical analyses of the features of 89 Pascal programs collected from the University of Colorado at Boulder and Tektronix, Inc. at Beverton, Oregon together with some discussion of the implications. For example, it is interesting to find that if-then-else statements (3.3%) make up about half of all the if-statements (6.7%) and are only slightly more frequent than case-statements (2.7%). A fragment of one of the tables will illustrate the data.

TABLE 5. Depth of Procedure Nesting

Nesting Depth	Number	Percent	Mean Percent	Std Dev
2	1439	59.59	83.90	27.59
3	545	22.57	10.20	17.55
4	260	10.77	3.07	7.81
5	93	3.85	0.94	4.14
6	57	2.36	0.50	2.68
7	13	0.54	0.14	0.70
8	6	0.25	0.07	0.32
9	2	0.08	0.03	0.19

The book is unlikely to be of interest to practising programmers; it may be of some interest to teachers and others interested in the design of programming languages, but its prime target must be the small collection of people involved in the implementation, validation and maintenance of compilers. For this group it can serve as confirmatory evidence for hunches, indications of where optimization effort may pay off, and suggestions for *virtual infinity* limits incorporated into compilers.

The book is well-written for this target group; the data is well-explained and broken down into its two main sub-groups as well. That different populations have different styles of programming is well-known; recent work by Dr Mike Rees showed some quite striking differences between practices at the University of Tasmania and at the University of Southampton.

A.H.J. Sale,
University of Tasmania

ENCARNACAO, J.L. (ed.) (1981): *Eurographics 81*, North-Holland, Amsterdam, 335 pp.

Since the European GKS (Graphics Kernel System) has become the likely basis for standardisation efforts in computer graphics, considerable interest exists in the state of computer graphics in Europe.

This volume contains the proceedings of the second Eurographics Conference, held at the Technische Hochschule, Darmstadt in the Federal Republic of Germany during September 1981. Its contents form as heterogeneous a collection of papers as could be expected from the general conference of the European version of SIGGRAPH.

Some thirty-one papers are included, grouped under thirteen broad headings, representing the sessions covered in the conference programme. These topics range from the usual: Hidden Surface Techniques, Graphics Standards, Curve Algorithms, Geometric Modelling, Image Processing, and Graphics Protocols, to the more esoteric (at least in title) sessions, Computer Graphics Prospective, and Economic and Social Implications of Computer Graphics.

The first section – Computer Graphics Prospective – is disappointing. An area with the growth of computer graphics should attract more comment on 'future direction' than the two papers in this section offer. The first, by Norman Badler and Robert Ellis, provides a history of SIGGRAPH (more retrospective than prospective). The second paper, however, by James Clark of Stanford University, does describe some interesting developments in hardware system design. Combining an understanding of integrated systems design and computer graphics, he illustrates how 'Geometry Engines' (simple, microprogrammed four component vector processors) can be organised in a pipeline to produce a Geometry System that can provide high-performance 2D and 3D graphics at microprocessor prices.

The papers on Hidden Line/Hidden Surface techniques both concentrate on computational efficiency. The paper by Hornung describes a method which computes the visible parts of a picture rather than excluding the invisible ones. The method compares favourably in computation time with Appel's Method and Encarnacao's Method. Schmitt's paper demonstrates, by the comparison of four algorithms, that the asymptotic time needed to eliminate hidden lines depends almost exclusively on properties of the scene rather than on the cleverness of the algorithm.

Two papers describe geometric modelling systems – CADLAN (Computer Aided Design Language) and GWB (Geometric Workbench). CADLAN contains some innovative procedures for constructing solids, such as being able to rotate a curve through space, while GWB uses a hybrid representation scheme which combines CSG and boundary representations. These systems only produce line drawings, in contrast to other systems such as GM Solid and PADL-2 which can produce shaded raster images. Another paper, by R. Hartwig, describes an interactive system for displaying irregularly spaced two dimensional data in a wide variety of ways.

There is a wide selection of applications descriptions, many of them built around GKS. This in itself is of interest to those of us who have assumed in our work the centrality of the CORE system in efforts to produce an international computer graphics standard. One paper, by Ten Hagen, does address the problem of standardisation. Unfortunately, by the time the proceedings have been published, we are all that much further along the path to a standard.

No doubt these proceedings do represent a valuable record of the state of European computer graphics. The presentation, non-uniform type styles, abstracts of interesting looking papers where the full text would have been valuable, are all by-products of the publication process of conference proceedings.

J.M. Hughes,
New South Wales Institute of Technology

BROOKES, C.H.P., GROUSE, P.J., JEFFERY, D.R., and LAWRENCE, M.J. *Information Systems Design*, Prentice-Hall, NJ, 468 pp., $26.95.

This book has been written to serve as a text for those who wish to become familiar with the tools, techniques and practice of systems design.

The authors have attempted to cover a very wide variety of related subdisciplines such as interview and other fact finding techniques, software design methodologies, database management systems, data communications technology, distributed systems, dialog specifications, networks, cryptographic systems, project selection and management; and all this in 468 pages.

The book contains a wealth of material, illustrated by a large number of case studies. Most topics are treated adequately and in sufficient detail. Where applicable, the most important alternative approaches are also briefly described. Each chapter is followed by a list of further references as well as some exercises.

The book is highly readable, although, due to the variety of topics, the overall structure is hard to find. Due to this lack of 'streamlining', we would not recommend its use as the primary text in an information systems course. However, it could be very useful as a reference text where students can find related and/or alternative material. The index is adequate and each section can be read by itself.

In summary, *Information Systems Design* contains much more than $27 worth of material (in order to obtain the same information one would usually expect to buy at least three books). Although it is, in our opinion, unsuitable as a first course text, it can serve as an excellent supplementary reference volume.

D. Vermeir,
University of Queensland

LAMOITER, J.P. (1982): *BASIC Exercises for the IBM Personal Computer*, Sybex Inc., Berkeley, California, soft cover, $A23.50.

This book sets out to teach introductory BASIC programming by working through programming exercises or problems. It is not about problem solving techniques, but rather it is more concerned with flowcharting and the translation of given algorithms into BASIC programs. The exercises are almost all numeric and include some less familiar examples as well as old favourites. Apart from the introductory part, the book is divided into chapters on the basis of field of application, there being only a weak connection from chapter to chapter. Within each chapter exercises are presented more or less in order of increasing difficulty. The general treatment of two specialised control constructs in Chapter 2 seems out of place, when more basic constructs such as loops are not discussed generally. The index is short and inadequate. There are various versions of BASIC, and while this book specifically deals with one, the author uses fairly standard constructs and does indicate changes which may be needed for other versions of the language.

Though elementary, the book implicitly assumes that the reader understands something about computers and programming. There is a brief explanation of some constructs, but others such as arrays, are used without any explanation. While it is a welcome change to have less attention focussed on language features, the reader will need a supplementary text or primer on BASIC.

The text is quite well written, and the diagrams are, on the whole, clear. As the programs and their results are reproduced directly from computer output, they contain very few typing errors (an exception is the misspelling of RETURN in one program). However, more errors occur in the text and the flowcharts; this is partly due to the use of a confusing type font in which uppercase I and lowercase l are the same and barely distinguishable from the digit one. Occasionally captions in the flowchart boxes are barely legible because of size reduction.

For each program described, the problem is first specified, any necessary mathematical analysis given and then the algorithm is presented either as a sequence of instructions in English (with variables and *gotos*) together with a flowchart, or directly as a flowchart; use is made both of macroscopic and detailed flowcharts, the latter with both variable names and control structures peculiar to BASIC. The flowcharts seem more complicated than the programs, often a whole page chart corresponding to a dozen lines of code. Indeed, one wonders if the flowcharts were drawn up before or after the programs were written; it is questionable whether they do help in understanding the programs.

Explanations, particularly of the simpler examples, are clear; the code corresponds as closely as possible to the flow charts, and care has been taken to keep the programs simple (including, apparently, the literal avoidance of *goto* statements); there is an unnecessary initialisation in one example and a number of peculiarly "BASIC" coding tricks are used. In some cases a top down approach is adopted and there is proper emphasis on using procedures; the physical and logical end of one program is a non-returning branch of a subroutine!

BASIC programs, despite the simple syntax, are comparatively hard to understand because of short names and primitive control constructs (in some versions). BASIC really needs more comments and care with layout than other high level languages, but this is discouraged by overheads of interpretation; the published programs are lacking in this regard. In some cases, descriptive

remarks have been added by typing on to the computer produced program texts.

The book uses interesting examples, adopts an adult approach and is designed for self instruction; it is not suitable as a school textbook or as an introduction to programming for academic or professional purposes. Reading this book confirms the view that flowcharting and BASIC, together or separately, are not appropriate notations for teaching modern programming. However, if one must use BASIC and one wishes to learn to code mainly numerical and mathematical algorithms as a hobby, then studying the exercises in this book would be useful.

B.P. Kidman,
University of Adelaide

YOVITS, M.C. (ed.) (1982): *Advances in Computers*, Volume 21, Academic Press, New York, 452 pp.

Advances in Computers has now been published more or less annually for more than 20 years. Each volume contains half a dozen tutorial or survey articles on topics of current and continuing interest. These are rather longer and more comprehensive than those in ACM's *Computing Surveys*, which, for example, published 494 pages devoted to 15 separate articles in Volume 13 (1981).

In the first article of this book, in "The Web of Computing: Computer Technology as Social Organization", Rob Kling and Walt Scacchi draw a dichotomy between *discrete-entity* and *web* models which can be used to analyse "the social consequences of socially complex computer applications and the difficulties encountered in their use". Initially their arguments seem a little strained, but the introduction of a few well-chosen examples adds much greater immediacy, and I found that the whole article made interesting and thoughtful reading.

The second article, "Computer Design and Description Languages" by Subrata Dasgupta surveys languages that "may be used for the design and description of computer hardware systems". Such languages have been studied since the 1950's, but the problems addressed are inherently difficult and progress has been uneven. Recently the prospect of implementing design aids to create complete microprocessors on a single silicon chip has refocussed attention in this area. It is no longer far-fetched to predict routine fabrication of special purpose microprocessors to suit particular applications. If and when this happens, some computer design and description languages will be important indeed.

The article "Microcomputer: Applications, Problems and Promise" by Robert C. Gammill is another well-written survey that will be useful reading for the non-initiated, but otherwise contains few surprises. The fourth article, "Query Optimization in Distributed Data Base Systems" by Giovanni M. Sacco and S. Bing Yao, covers an area of active current research interest with important practical ramifications. It assumes a reasonable level of prior knowledge of readers and describes a number of important algorithms. The most recent reference quoted is early 1981.

For anyone whose impressions of chemistry are predominantly those of high school "wet chemistry" in test tubes, the degree to which computer methods now permeate modern chemistry may come as a surprise. The fifth article, "Computers in Chemistry" by Peter Lykos, lists many, many separate areas of application, and the author's beliefs are epitomised by the following quotation:

> "There is no foreseeable limit to the complexity of the systems chemists can model. There is no foreseeable scientific computer that chemistry modelers cannot exploit fully."

The article is a wide-ranging and presumably authoritative introduction to its subject. It would be highly recommended reading for those with a general interest in the topic, or who are involved in teaching about computing to chemistry students.

Library automation is a project of long standing, but one which has suddenly matured under the pressures of computer evolution, the information explosion and the deflation of library operating budgets. Perhaps library administrations have been far too unadventurous: for example, the implementation of automated circulation control systems – by no means a wide-spread phenomenon, even today – may just be the small tip of a very large iceberg. James Rush, in the final article "Library Automation Systems and Networks", is just as thorough and thought-provoking as the earlier authors.

This book is recommended for reading for all computer professionals, and certainly for acquisition by libraries. It could form excellent background material for any group of computerists, who, struggling to keep up in this burgeoning field, wish to organise a regular discussion group on topics of current interest.

J. Lions,
University of New South Wales

DAVIS, W.S. (1983): *Operating Systems: A Systematic View*, Addison-Wesley Publishing Company, Reading, Mass. Second Edition, 520 pp., $A24.95.

It seems that the computer world is still sharply divided between those that know and care about IBM's operating systems, and those that don't. I cannot say what most readers would expect from a book with this title, but I expected a little more than another recital of the gothic spendours of DOS, OS and all their progeny, not to mention JCL. To show how widely based this book really is, let me quote from p. 300:

> "It's not a new idea. Back in the early 1960's (prehistory as far as computers are concerned), Burroughs had a fully operational system (they didn't call it 'virtual') for their 5000 computer series. UNIVAC, CDC and GE/Honeywell all had it during the 1960's. RCA was the first to use the term 'virtual memory' in 1970. The key date in the development of the virtual memory concept was, however, August 2, 1972, when IBM announced its System/370 series of computers; for the first time *the* major computer manufacturer had decided to make the virtual memory concept a key element in a full line of computers."

Readers will be fascinated to know that that paragraph constitutes the entire reference in this book to the efforts of Burroughs, CDC, GE, Honeywell, RCA and UNIVAC. A systematic study indeed! Not even a fig for the likes of Atlas and Multics. However, not all is completely lost: there is a half-hearted attempt to introduce CP/M and the HP 3000 series, and UNIX scores a one-line mention as "an operating system to support certain minicomputers". (Didn't anyone tell the author . . .)

Clearly, this is not a book for real system programmers, and the title is quite misleading. The question remains whom might the book serve? It is reasonably clearly written, and the use of IBM acronyms is not over-powering. On the other hand, the level of detail is not high, and anyone who actually has to use IBM systems will need all the extra manuals he or she can find. It might be a useful introduction for a manager in an IBM installation who has no previous familiarity with operating systems, or to people with no option but to learn about operating systems via IBM systems or to people who are just fascinated by exercises such as:

> "Code a DD statement DCB parameter for a magnetic tape file holding fixed-length blocked records – logical records are 50 characters in length and the blocking factor is 50. It's a 1600-bpi tape."

This book, with its fixation on the concepts and practices of the 1960's, is hardly recommended reading for the 1980's. Forget it.

J. Lions

MICHAELSON, H.B. (1982): *How to Write and Publish Engineering Papers and Reports*, ISI Press, University City Science Center, Philadelphia, PA 19104, USA, 158 pp. (soft cover) $US14.95 plus $3.00 postage.

The author has had many years experience both as a writer of technical papers and as associate editor of the IBM Journal of Research and Development. While possibly not the last word on this subject, this book is full of good advice and insights into writing technical papers. The first chapter is entitled "How to Define Quality in Enginering Manuscripts" and begins:

> "An engineer's work is never quite complete until he or she has described what was accomplished, by writing a technical report for an engineering organization or submitting a paper for journal publication. In either case, the quality of the manuscript reflects on the character of the author's work and reputation."

The book goes on to describe how to recognise quality in this context; how to match the paper's objectives with reader interests; how to plan the paper or report; how to go about creating it; how to write an abstract, an introduction, an effective concluding section; how to choose and prepare illustrations, drawings and tables for maximum impact; how to cite references and to compile bibliographies; and finally, how to complete one's manuscript and submit it successfully for publication. It also contains many hints as to how to achieve proper emphasis, to avoid strategic errors, and to increase one's own writing productivity.

Writing technical papers is a skill which is not unfamiliar to most of us, but which is practised well by relatively few. There will be few readers who cannot find much useful advice in this book. Its expense could easily be recouped when you write your next paper.

J. Lions,
University of New South Wales

SPECIAL ISSUE ON

SOCIAL CONSEQUENCES OF COMPUTING TECHNOLOGY

The *Australian Computer Journal* will publish a special issue on "Social Consequences of Computing Technology" in November 1983. Research papers, tutorial articles and industry case studies on all aspects of the subject will be welcome, and both full papers and short communications will be considered.

Prospective authors should write as soon as possible to:

Ashley W. Goldsworthy,
P.O. Box 554,
Fortitude Valley, Qld. 4006

to notify him of their intention to submit material for the issue and provide a brief summary of their intended contribution.

In order to allow adequate time for refereeing and editorial review, complete manuscripts will be required no later than 15 June 1983.

Papers should be prepared in accordance with the guidelines published in the November 1982 issue of the Journal. Authors are requested to pay particular regard to the Journal's prefered style for references.

(Continued from page I)

In the automatic model the working height is adjusted by means of a code card. The ideal position for each user is determined in advance and recorded on the code card.

When the user inserts the card into the relevant slot, the two leaves immediately adjust themselves automatically.

If required the computer tables can be provided with two side leaves for easy combination of other furniture. The lifting mechanism is also available separately.

Manufactured by Emmein B.V., Postbus 2, 7800 AA Emmen (Netherlands). Telex: 30087.

MAINFRAME DEVELOPMENT CENTRE OPENED

ICL's new $36 million mainframe development centre at West Gorton in Manchester, England, has been officially opened by Sir Christophor Laidlaw, chairman of the ICL Group. All design work for extending the existing range of ICL 2900 series computers and for succeeding generations will take place at the centre. Other activities include the design of the world's largest logic chips, which will be manufactured by Fujitsu in Japan, for use in extensions to the 2900 series.

The West Gorton site comprises 13,283 square metres and houses Europe's largest computer hall of 2,165 square metres, which in itself represents an investment by the company of $17 million in computer equipment. A further $19 million has been spent on the buildings. Over 1,000 people are employed there with 80 per cent of the professional staff holding university degrees.

NEW 1983/4 DICK SMITH WHOLESALE ELECTRONICS ENTHUSIAST'S CATALOGUE

Dick Smith Electronics has announced the release of its latest Electronics Enthusiast's Catalogue. This 150 page catalogue is packed with new products which are certain to trigger the imagination of the professional and hobbyist alike.

The 36 Dick Smith Electronics stores throughout Australia stock nearly 3,500 items ranging from components to Hi-Fi, alarms to amateur radio and telephone products to video games and home computer systems. The catalogue lists them all, with thousands of detailed illustrations and photographs, many in full colour, and a completely updated and revised data section.

NEW FIBRE OPTICS CAPABILITY FROM CENTRE INDUSTRIES

An exclusive Australasian distributorship for an advanced Canadian fibre optic communications system has been secured by the Sydney-based telecommunications and electronics specialist, Centre Industries.

According to a company spokesman, the deal with Ottawa-based Foundation Electronic Instruments Inc. will strengthen the diversification effort of the Australian company, which earlier this year made a successful step into manufacturing hardware for the personal computer market.

Centre Industries has become a significant supplier of equipment to Telecom, and the new fibre optics capability will develop the company's position in the expanding high-technology communications market.

Whilst initial customers are expected to be located in the broadcast industry, it is anticipated that the fibre optic technology will be widely employed in future telecommunications transmission activity, with substantial cost and efficiency improvements over present systems.

Initial shipments of fibre optics products are expected in Australia within the next few years.

PIRATE-PROOF COMPUTER PROGRAMMES

Computer software packages that cannot be copied have been developed by a British company.

Claimed to be the first "pirate-proof" system available in the world, Parwest of Chippenham, Wiltshire, south west England, says a numerical code system will protect the range of business and accounting software which it plans to offer for rental.

Managing Director, Mr. Keith Park, a computer business systems analyst, said the system was developed as a private hobby interest over the past three years. It allows software to be rented out without the danger of it being copied by the renter — a practice which has stopped the development of this market and which is said by Britain's Computer Software Association to be a growing problem.

It is only because Mr. Park is confident that his code is unlikely to be broken that he can consider renting out software for short periods. People renting the programmes can assess them before deciding if they need them.

Mr. Park said the security is built into the programme. The data files contain a multi-dimensional numerical matrix which runs according to a set pattern. This sequence runs out at pre-determined intervals. If the programme is to continue, the user has to obtain the next few numbers by telephoning a Parwest office.

Without the extra numbers, which will be given only to bona fide customers, the programme will lock and cannot be used.

Mr Park added that systems have been in use with three customers for the past four months.

IBM APPLICATIONS WITH NEW 88CARD FOR APPLE COMPUTERS

Personal Computer Products, Inc. (PCPI) a California-based microcomputer board manufacturer recently announced their second co-processing board in less than a year.

The PCPI 88CARD is plug compatible with Apple II, II plus, IIe, and other Apple compatible microcomputers. The 88CARD comes standard with 64k of on-board memory. Add this to a 64K Apple II and the 16-bit 8088 microprocessor can address 128k bytes. This additional memory space allows users to run larger, more sophisticated programs adding power to the Apple microcomputer.

"Presently, the primary function of the 88CARD is as a developer's tool", stated Ed Savarese, President of PCPI. "With over 750,000 Apple computers in the market today, our 88CARD opens the door to allow developers to use their Apple computers to write application software under MS-DOS for the IBM PC."

According to Savarese, the day is not far off where software houses will provide Apple formatted IBM programs that will run under the MS-DOS operating system. Software houses are encouraged to contact PCPI to have their particular software formatted for the 88CARD.

The 88CARD comes with MS-DOS and MBASIC. CP/M-86 will be an option.

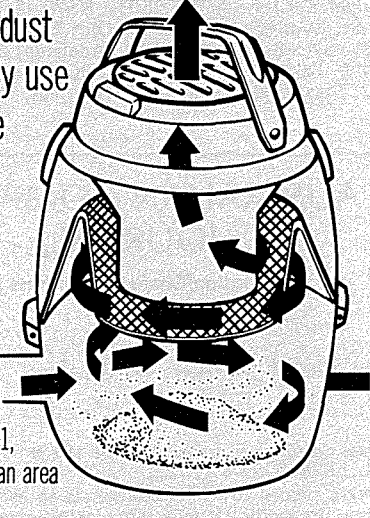
For further details on the 88CARD contact Personal Computer Products, Inc., 16776 Bernardo Center Drive, San Diego, Ca. 92128. Telephone (619) 485-8411. Telex: 4992939 PCPI SDG.

HOW NILFISK CAN KEEP YOUR COMPUTER CLEAN.

Even with today's computers, a dust free environment is still advisable, for both your staff and your office equipment.

But every day, people bring in dust from outside, printers create paper dust, carpets produce fluff and then the air conditioning system helps circulate all this throughout your whole office complex.

Nilfisk can keep your office, and most importantly your disk drives, clean and efficient with their GS80 vacuum cleaner. Its unique cyclonic action makes it quieter and more powerful than any other conventional barrel or upright vacuum cleaner. Its filtration system can trap 99.9% of all dust particles larger than .03 microns–that's 0.00003mm! You can safely use your Nilfisk vacuum cleaner whilst your computer is in use, because the Nilfisk creates no electronic interference. Nilfisk will supply a wide range of attachments for cleaning all types of surfaces around your office. So, don't put your staff or equipment at risk –clean up with Nilfisk.



Unique cyclonic action and filtration system of Nilfisk's GS 80.

For more information either fill in this coupon and mail it to: Nilfisk of Australia P/L, P.O Box 21, CONCORD WEST, N.S.W., 2138 or phone 736 1244. For calls from outside the Sydney metropolitan area phone 008-22 4326 at the cost of only a local call.

Name: _____ Company: _____

Address: _____ Phone Number: _____

CLEAN UP WITH NILFISK

H&H/NOA/18/83.

The Australian Computer Journal is an official publication of the Australian Computer Society Incorporated.

OFFICE BEARERS: *President:* A.W. Goldsworthy, *Vice-Presidents:* B.M. Hannelly, A.H.J. Sale; *Immediate past president:* G.E. Wastie; *National treasurer:* R.A. Hamilton; *Chief executive officer:* R. Shelley, PO Box N26, Grosvenor Street, Sydney, 2000, telephone (02) 267-5725.

EDITORIAL COMMITTEE: *Editor:* J. Lions, University of New South Wales. *Associate Editors:* J.M. Bennett, P.C. Poole, A.Y. Montgomery, C.K. Yuen.

SUBSCRIPTIONS: The annual subscription is $20.00. All subscriptions to the Journal are payable in advance and should be sent *(in Australian currency)* to the Australian Computer Society Inc., PO Box N26, Grosvenor Street, Sydney, 2000. A subscription form may be found below.

PRICE TO NON-MEMBERS: There are now four issues per annum. The price of individual copies of back issues still available is $2.00. Some are already out of print. Issues for the current year are available at $5.00 per copy. All of these may be obtained from the National Secretariat, PO Box N26, Grosvenor Street, Sydney, 2000. No trade discounts are given, and agents should recover their own handling charges.

MEMBERS: The current issue of the Journal is supplied to personal members and to Corresponding Institutions. A member joining partway through a calendar year is entitled to receive one copy of each issue of the Journal published earlier in that calendar year. Back numbers are supplied to members while supplies last, for a charge of $2.00 per copy. To ensure receipt of all issues, members should advise the Branch Honorary Secretary concerned, or the National Secretariat, promptly, of any change of address.

MEMBERSHIP: Membership of the Society is via a Branch. Branches are autonomous in local matters, and may charge different membership subscriptions. Information may be obtained from the following Branch Honorary Secretaries. Canberra: PO Box 446, Canberra City, ACT, 2601. NSW: Science House, 35-43 Clarence St, Sydney, NSW, 2000. Qld: Box 1484, GPO, Brisbane, Qld, 4001. SA: Box 2423, GPO, Adelaide, SA, 5001. WA: Box F320, GPO, Perth, WA, 6001. Vic: PO Box 98, East Melbourne, Vic, 3002. Tas: PO Box 216, Sandy Bay, Tas, 7005.

AUSTRALIAN COMPUTER JOURNAL

Subscription/Change of Address Form

Name . . . . . .

Current Address . . . . . .

. . . . . .

☐ Please enrol me as subscriber for 1983. I enclose a cheque or bank draft for $A20.00.

☐ Please record my new address as shown above. I attach below the mailing label for the last received issue.

ATTACH LABEL HERE

Send all correspondence regarding subscriptions to PO Box N26, Grosvenor Street, Sydney, 2000, Australia. Photocopies of this form are acceptable.

CONTRIBUTIONS: All material for publication should be sent to: Associate Professor J. Lions, Editor, Australian Computer Journal, Department of Computer Science, University of New South Wales, Kensington, NSW 2033. Prospective authors may wish to consult manuscript preparation guidelines published in the November 1982 issue. The paragraphs below briefly summarise the essential details.

Types of Material: Four regular categories of material are published: Papers, Short Communications, Letters to the Editor and Book Reviews. Generally speaking, a paper will discuss significant new results of computing research and development, or provide a comprehensive summary of existing computing knowledge with the aim of broadening the outlook of Journal readers, or describe important computing experience or insight. Short Communications are concise discussions of computing research or application. A letter to the Editor will briefly comment on material previously appearing in the Journal or discuss a computing topic of current interest. Descriptions of new software packages are also published to facilitate free distribution.

Refereeing: Papers and Short Communications are accepted if recommended by anonymous referees, Letters are published at the discretion of the Editor, and Book Reviews are written at the Editor's invitation upon receipt of review copies of published books. All accepted contributions may be subject to minor modifications to ensure uniformity of style. Referees may suggest major revisions to be performed by the author.

Proofs and Reprints: Page proofs of Papers and Short Communications are sent to the authors for correction prior to publication. Fifty copies of reprints will be supplied to authors without charge. Reprints of individual papers may be purchased from Associated Business Publications, PO Box 440, Broadway, NSW, 2007. Microfilm reprints are available from University Microfilms International, Ann Arbor/London.

Format: Papers, Short Communications and Book Reviews should be typed in double spacing on A4 size paper, with 2.5cm margins on all four sides. The original, plus two clear bond-paper copies, should be submitted. References should be cited in standard Journal form, and generally diagrams should be ink-drawn on tracing paper or board with stencil or Letraset lettering. Papers and Short Communications should have a brief Abstract, Key word list and CR categories on the leading page, with authors' affiliations as a footnote. The authors of an accepted paper will be asked to supply a brief biographical note for publication with the paper.

This Journal is Abstracted or Reviewed by the following services:

Publisher	Service
ACM	Bibliography and Subject Index of Current Computing Literature.
ACM	Computing Reviews.
AMS	Mathematical Reviews.
CSA	Computer and Information Systems Abstracts. Data Processing Digest.
ENGINEERING INDEX INC.	Engineering Index.
INSPEC	Computer and Control Abstracts.
INSPEC	Electrical and Electronic Abstracts.
ISI	Current Contents/CompuMath
ISI	CompuMath Citation Index.
SPRINGER-VERLAG	Zentralblatt fur Mathematick und ihre Grenzgebiete.

Production Management: Associated Business Publications, Room 104, 3 Smail Street, Ultimo, NSW 2007 (PO Box 440, Broadway, NSW 2007). Tel: 212-2780, 212-3780.
All advertising enquiries should be referred to the above address.

Printed by: Publicity Press (NSW) Pty Ltd, 66 O'Riordan Street, Alexandria, NSW 2015.